FORMAL TECHNIQUES FOR NETWORKED AND DISTRIBUTED SYSTEMS
IFIP - The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP’s aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

IFIP’s mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP’s events range from an international congress to local seminars, but the most important are:

- The IFIP World Computer Congress, held every second year;
- open conferences;
- working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly. National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.
Contents

Preface ix
Program Committee and Reviewers xi

Part One
FORMAL METHODS IN SOFTWARE DEVELOPMENT I

1. Automated Derivation of ILP Implementations from SDL Specifications
   S. Twarok, P. Langendoerfer and H. Koenig 3

2. Stepwise Design with Message Sequence Charts
   F. Khendek, S. Bourduas and D. Vincent 19

3. Formal Synthesis and Control of Soft Embedded Real-Time Systems
   P.-A. Hsiung 35

Part Two
DISTRIBUTED SYSTEMS TESTING

   C. Viho, S. Barbin and L. Tanguy 53

5. Distributed Test using Logical Clock
   Y. J. Choi, H. Y. Youn, S. Seol and S. J. Yoo 69

6. Diagnosing Multiple Faults in Communicating Finite State Machines
   K. El-Fakih, N. Yevtushenko and G. v. Bochmann 85

7. From Active to Passive : Progress in Testing of Internet Routing Protocols
   J. Wu, Y. Zhao and X. Yin 101

Part Three
TIMED AUTOMATA

8. Time and Action Lock Freedom Properties for Timed Automata
   H. Bowman 119
9. Compiling Real-Time Scenarios into a Timed Automaton
   A. Salah, R. Dssouli and G. Lapalme

10. Deriving Parameter Conditions for Periodic Timed Automata
    Satisfying Real-Time Temporal Logic Formulas
    A. Nakata and T. Higashino

Part Four
PROCESS ALGEBRA

11. PAMR: A Process Algebra for the Management of
    Resources in Concurrent Systems
    M. Núñez and I. Rodriguez

12. A Symbolic Semantics and Bisimulation for Full LOTOS
    M. Calder and C. Shankland

13. Implementing a Modal Logic over Data and Processes using XTL
    J. Bryans and C. Shankland

Part Five
APPLICATIONS OF VERIFICATION

14. Formal Verification of Peephole Optimizations in
    Asynchronous Circuits
    X. Kong and R. Negulescu

15. Symbolic Verification of Complex Real-Time Systems with
    Clock-Restricition Diagram
    F. Wang

16. Verifying a Sliding Window Protocol using PVS
    V. Rusu

Part Six
TEST SEQUENCE DERIVATION

17. Test Sequence Selection
    D. Lee and R. Hao

18. Executable Test Sequence for the Protocol Data Flow Property
    W.-H. Chen
19. A Method to Generate Conformance Test Sequences for FSM with Timer System Call
   T. Mori, K. Tokuda, H. Tada, M. Higuchi and T Higashino

Part Seven
FORMAL METHODS IN SOFTWARE DEVELOPMENT II

20. A Tool for Generating Specifications from a Family of Formal Requirements
   J. Bredereke
   319

21. Patterns and Rules for Behavioural Subtyping
   H. Wehrheim
   335

Part Eight
THEORIES OF VERIFICATION

22. Verification of Dense Time Properties using Theories of Untimed Process Algebra
   M. Luukkainen
   353

   U. Ultes-Nitsche and S. St James
   369

24. SVL: A Scripting Language for Compositional Verification
   H. Garavel and F. Lang
   377

Part Nine
INVITED PAPERS

25. On Formal Techniques in Protocol Engineering – Example Challenges
   D. Bjørner
   395

26. A PKI-Based End-to-End Secure Infrastructure for Mobile E-Commerce
   T.-W. Cheung and S. T. Chanson
   421

27. A Family of Resource-Bound Real-Time Process Algebras
   443

28. Survivability Analysis of Networked Systems
   J. M. Wing
   459
This page intentionally left blank.
Preface

FORTE 2001, formerly FORTE/PSTV conference, is a combined conference of FORTE (Formal Description Techniques for Distributed Systems and Communication Protocols) and PSTV (Protocol Specification, Testing and Verification) conferences. This year the conference has a new name FORTE (Formal Techniques for Networked and Distributed Systems). The previous FORTE began in 1989 and the PSTV conference in 1981. Therefore the new FORTE conference actually has a long history of 21 years.

The purpose of this conference is to introduce theories and formal techniques applicable to various engineering stages of networked and distributed systems and to share applications and experiences of them. This FORTE 2001 conference proceedings contains 24 refereed papers and 4 invited papers on the subjects. We regret that many good papers submitted could not be published in this volume due to the lack of space.

FORTE 2001 was organized under the auspices of IFIP WG 6.1 by Information and Communications University of Korea. It was financially supported by Ministry of Information and Communication of Korea.

We would like to thank every author who submitted a paper to FORTE 2001 and thank the reviewers who generously spent their time on reviewing. Special thanks are due to the reviewers who kindly conducted additional reviews for rigorous review process within a very short time frame. We would like to thank Prof. Guy Leduc, the chairman of IFIP WG 6.1, who made valuable suggestions and shared his experiences for conference organization.

This year we have seen exceptionally concerted efforts of the program committee to make FORTE 2001 a successful conference. We thank each one of the program committee for their contribution and cooperation. The enthusiasm and dedication the program committee showed has made us believe that FORTE will remain a prestigious conference with quality and distinction for a long time in the future.

Myungchul Kim
Byoungmoon Chin
Sungwon Kang
Danhyung Lee

Cheju Island, Korea
This page intentionally left blank.
Program Committee and Reviewers

Program Committee Co-Chairmen

Myungchul Kim (ICU, Korea)
Byoungmoon Chin (TTA, Korea)
Sungwon Kang (KT, Korea)
Danhyung Lee (KIPA, Korea)

Program Committee

G. v. Bochmann (Univ. of Ottawa, Canada)
T. Bolognesi (IEI, Italy)
H. Bowman (Univ. of Kent, UK)
E. Brinksma (Univ. of Twente, Netherlands)
S. Budkowski (INT, France)
A. Cavalli (INT, France)
S. D. Cha (KAIST, Korea)
S. Chanson (Hong Kong Univ. of Science and Technology, China)
E. H. Choi (Korea Telecom, Korea)
J. Y. Choi (Korea Univ., Korea)
J. P. Courtiat (LAAS-CNRS, France)
R. Dssouli (Univ. of Montreal, Canada)
D. Frutos-Escrig (Universidad Complutense - Madrid, Spain)
S. Fischer (International Univ., Germany)
R. Groz (France Telecom R&D, France)
R. Gotzhein (Univ. of Kaiserslautern, Germany)
T. Higashino (Osaka Univ., Japan)
D. Hogrefe (Univ. of Luebeck, Germany)
G. J. Holzmann (Bell Labs, USA)
J. K. Kim (ETRI, Korea)
S. U. Kim (Pukyung Univ., Korea)
H. Koenig (BTU, Germany)
R. Lai (La Trobe Univ., Australia)
D. Latella (C.N.R., 1st. CNUCE, Italy)
G. Leduc (Univ. of Liege, Belgium)
D. Lee (Bell Labs, USA)
D. I. Lee (KJIST, Korea)
J. Y. Lee (Yonsei Univ., Korea)
L. Logrippo (Univ. of Ottawa, Canada)
E. Najm (ENST, France)
D. Peled (Bell Labs, USA)
A. Petrenko (CRIM, Canada)
O. Rafiq (Univ. of Pau, France)
K. Tarnay (Nokia, Hungary)
R. Tenney (Univ. of Massachusetts, USA)
K. Turner (Univ. of Stirling, UK)
U. Ultes-Nitsche (Univ. of Southampton, UK)
S. Vuong (Univ. of British Columbia, Canada)
J. Wu (Tsinghua Univ., China)
N. Yevtushenko (Tomsk Univ., Russia)

Further Reviewers

D. Amyot          P. Gradit          J.- F. Monin
R. Andrade        S. Gruner          M. Nakamura
A. Bailly         R. Hao             A. Nakata
A. Bertolino      M. Heiner          H. Neukirchen
S. Boroday        H. Hermanns       S. Prokopenko
L. Cacciari       M. Higuchi        R. Pugliese
O. Catrina        I. Hwang           Y.- M. Quemener
D. Chen           A. Idoue           C. Rinderknecht
R. G. Clark       P. Janowski       A. Salah
M. Ebner          T. Kato            J. Sincennes
A. En-Nouaary     R. Langerak       A. Verdejo
A. Fantechi       C. Liu             F. Vernadat
A. Fontaine       L. Llana           H. Yamaguchi
N. De Francesco   S. Maag            G. Yang
C. Gervy          E. Magill          H. Yokota
S. Gnesi          J.’ C. Maldonado   J. Zhu
J. Grabowski
Part One

Formal Methods in Software Development I
This page intentionally left blank.
AUTOMATED DERIVATION OF ILP IMPLEMENTATIONS FROM SDL SPECIFICATIONS

Sven Twarok¹, Peter Langendoerfer², Hartmut Koenig

Brandenburg University of Technology at Cottbus, Department of Computer Science, PF 101344, 03013 Cottbus, Germany, Tel./Fax: +49-355-69-2236 / 2127, email: twarok@unilab.de, langendoerfer@ihp-microelectronics.com, koenig@informatik.tu-cottbus.de

Abstract

In this paper we present a mapping strategy for an important manual protocol implementation technique: the Integrated Layer Processing (ILP) which is used for protocol stack implementations with complex data manipulation operations. The Integrated Layer Processing was implemented beside the server and the activity thread model as third code generation procedure in the configurable SDL compiler COCOS. We describe the chosen transformation principle and discuss in particular the semantic problems which have to be solved for this mapping. Finally we present first measurements which show the effect of the transformation. Performance gains up to 20 per cent compared to COCOS server implementations and of about 300 per cent compared to the SDT Cadvanced tool were measured.

Keywords: Formal description techniques, integrated layer processing, automated protocol implementation, configurable FDT compiler, SDL

1. MOTIVATION

The increasing range of specialized communication solutions, in particular in mobile communication [Lang01], will increase the importance of the deployment of automatically generated code in real-life protocol implementations. Automated code generation can make the implementation process rather efficient and can fast adapt to changes in the design. The

¹ Sven Twarok is now with UniLab Software AG Braunschweig
² Peter Langendoerfer is now with IHP Microelectronics Frankfurt (Oder)
acceptance of automated implementation techniques depends on the efficiency of the generated code. Automated implementation techniques will be only then applied if the efficiency of the generated code comes close to that of manual implementations. The research of the last 10 years in this area has shown that there is still a large potential for efficiency increase by introducing optimizing techniques in the code generation process as discussed in [Lang99]. A remarkable efficiency gain can be achieved by adapting implementation techniques which are used for manual protocol implementations [Koen00]. In particular two approaches have been pursued:

- the integration of advanced implementation techniques and
- a flexible adaption to the implementation context.

The solution proposed for this is that of a configurable FDT compiler. The SDL compiler COCOS (Configurable Compiler for SDL), developed in our group, is an example for this approach. It possesses a variable runtime system which can be configured according to the requirements of the implementation context and the applied implementation technique. COCOS supports different implementation techniques as the server model, activity threads, the integrated layer processing, and techniques for avoiding data copying within a protocol stack. The basic principles of COCOS and the applied mapping strategies were presented in [Lang99]. In this paper we describe the generation of implementations according to the Integrated Layer Processing technique which has not been reported so far.

The Integrated Layer Processing (ILP) is an efficient implementation technique for protocol stacks with complex data manipulation operations [Clar90]. It in particular aims at protocol implementations with high data throughput requirements such as multimedia transmissions. The Integrated Layer Processing is especially applied to implement applications which follow the Application Level Framing principle. The Application Level Framing (ALF) approach supports a layer-independent processing of the data units. It aims among others at an application oriented definition of the data units, so called application data units, e.g. video frames, and the avoidance of protocol functions like fragmentation and multiplexing. The idea of the Integrated Layer Processing is to reduce memory accesses which may become a performance bottleneck in high performance communication when handling complex data units. This is achieved by a combined execution of complex data manipulation operations (encryption, decryption, checksum calculation, syntax transformations) in one or two memory access cycles independently of the layer the protocols belong to.

In this paper we show how ILP implementations can be automatically derived from SDL specifications. In Section 2 we give a short introduction to the Integrated Layer Processing implementation technique. Section 3 gives a short overview of related work. Section 4 describes the applied mapping
principle for SDL. Section 5 discusses semantic constraints. Section 6 sketches the code generation process. In Section 7 we present measurements to show the effect of the approach. The concluding remarks give an outlook on further required research steps.

2. **PRINCIPLE OF INTEGRATED LAYER PROCESSING**

The Integrated Layer Processing (ILP) represents an advanced protocol implementation technique. It was proposed by Clark and Tennenhouse [Clar90]. It is dedicated to the efficient implementation of data manipulation operations in protocols with a high data throughput. Typical examples for such data manipulation operations are the calculation of the checksum for error control, data encryption/decryption, data marshalling/unmarshalling and the compression/decompression of data streams. The approach is based on the observation that the execution of data manipulation operations over a protocol data unit (PDU) consumes more time for reading and writing of the data from and into the memory than for the proper execution of the operation. Since these data manipulation operations due to their functionality belong to different protocol layers many memory accesses are required in a traditional protocol stack implementation. Memory access times decrease at a slower rate compared to increase of the processor execution speed. For that reason memory access has become a bottleneck in high performance communication.

The Integrated Layer Processing aims at reducing the number of memory accesses by a combined execution of all data manipulation operations (DMOs) of a protocol stack. The DMOs are grouped into the so called ILP loop (see Figure 1). The DMOs can belong to different layers. They can even be located in different address spaces. The execution of the ILP loop does not influence the protocol execution, because the ILP loop combines operations which otherwise would be executed sequentially when the PDUs pass the different protocol layers. For execution, the DMOs are nested. They pass the data from one operation to the other whereby only fragments of the PDUs are processed each time. The DMOs aim at possibly small data fragments which fit into the register set of the processor. Thus only one read and one write operation per PDU are required for executing the ILP loop. The DMOs often use a different number of bytes as basis for their calculation. The calculation of the checksum is processed byte-wise, while an RSA encryption needs 16 byte blocks. To adjust the PDU fragments in an ILP loop word filters were proposed in [Abbo93] which concatenates bytes according to the length needed for the next DMO.
Figure 1: Principle of the Integrated Layer Processing

The ILP approach aims at the execution of the correct protocol paths, also called common paths [Leue96]. If an protocol error occurs in one of the protocols another execution path must be pursued to handle this error. In this case the results of the already executed DMOs of the higher layers are not needed any more and will be discarded. Low protocol error rates do not much influence the performance gain of the approach. If the error rate increases other implementation techniques will be more appropriate. We also discuss this effect later in this paper.

The Integrated Layer Processing can bring remarkable performance gains. For simple data manipulation operations like the calculation of the checksum, performance gains of up to 50 per cent were observed [Clar90], [Abbo93]. Further experiments revealed that the performance increase depends on the complexity of the data manipulation operations. It is less for more complex operations [Brau95]. In [Ahlg96] it was shown that ILP can even in part lead to efficiency losses compared to traditional implementation techniques if all data and state information of the data manipulation operation to execute cannot be stored in the register set of the processor. These data are moved to the cache or the memory. If the data are held in the cache the decrease may not be that strong but if the required information is neither found in the registers nor in the cache the performance loss may be dramatic. The deployment of the technique therefore requires a careful preparation to use its benefits.
3. RELATED WORK

The use of ILP has been investigated for formal description techniques (FDTs). For SDL, an approach for deriving the common paths was proposed in [Leue96], but no mechanisms for error recovery were discussed. The treatment of semantic constraints arising, for instance, from transitions with save statements also has not been solved. To our knowledge a compiler using ILP for one of the standardized FDTs has not been implemented yet. Automated derivations of ILP implementations from formal description techniques have been only reported for Esterel so far [Brau96], but the paper scarcely gives hints on the handling of semantic constraints. It is reported that the generated code achieves the efficiency of hand-coded implementations. Esterel though is more implementation oriented compared to the standardized FDTs. The problems for applying the technique to SDL arise from semantic constraints like the asynchronous communication between the processes. A practicable solution for SDL was proposed in [Twar00]. It is presented in the following.

4. MAPPING ON ILP IMPLEMENTATIONS

The mapping of SDL specifications on ILP implementations is not straight-forward. There are several problems to solve. First it is necessary to have an SDL specification of several protocol layers or of the whole protocol stack, respectively. We assume in the following that this specification is given. Furthermore, data manipulation operations are usually not entirely specified using SDL statements. In addition, SDL exchanges only complete signals between processes. The specification therefore does not provide sufficient support for a byte-wise exchange of the signals between DMOs. The compiler cannot always nest the DMOs. In order to use nevertheless the benefits of ILP implementations we distinguish two kinds of ILP loops:

- the simple ILP loop and
- the advanced ILP loop.

The loops differ in the nesting degree of the DMOs, but both variants lead to optimized implementations.

Simple ILP loop

In simple ILP loops the DMOs are not nested. Instead of this they are mapped into a common procedure and executed sequentially. Figure 2 shows the principle. The transitions which are involved in the loop remain preserved in the implementation. The PDUs of the different layers can be
uniquely identified. Protocol sequences which cannot be mapped on this loop are implemented in the usual way. No additional measures are applied.

Figure 2: Structure of the simple ILP loop

The performance gain of simple ILP loops results from the avoidance of data copy operations between the DMOs. Besides no runtime support is needed for the process change. This is implicitly done by the ILP loop. The overhead needed for the signal exchange and the execution of the respective operation is avoided.

Advanced ILP loop

The advanced ILP loop fully implements the ILP concept as proposed in [Clar90]. The DMOs are nested and enforce a byte-wise processing of the data units. Because of this the PDUs of the different layers cannot be identified any more. All decisions concerning the correct transmission of the PDUs have to be delayed until the processing of the data unit is completed (see Figure 3). After that the decisions are evaluated. If the PDUs were correctly transmitted the respective protocol state is updated and the output statements are executed. For this, the transitions belonging to the loop must be restructured. This can be done automatically based on the following rules:

1. Move all decisions which influence the protocol flow behind the DMO execution,
2. Postpone outputs until these decisions are made,
3. Suppress all outputs if a protocol error occurs and trigger the error handling procedure,
4. Include word filters if the DMOs use different data formats (see below).
The performance gain of the *advanced ILP loop* is of course higher than that of the *simple loop* because only registers are used for the data exchange between DMOs.

**Specification of ILP loops and word filters**

Another important problem is the specification of the DMOs which belong to a certain ILP loop. The SDL specification does not contain any information on this. This information has to be included into the implementation specification which is the starting point for automated code generation. The implementation specification refines the protocol specification for the given target systems and includes all information for automatically deriving an implementation. The reasons for the use of an implementation specification are given in [Lang99], [Koen00].

**Figure 3: Structure of the advanced ILP loop**

In our approach implementation oriented information is included by means of iSDL statements [Lang99]. iSDL is an implementation oriented annotation to control the code generation process. The statements are included as comments into the SDL specification and translated together with the specification. iSDL provides statements for specifying ILP loops and word filters.

An ILP loop is specified by means of the statements *define simple/advanced ILP loop*, e.g.

```plaintext
/!*{$ define simple ILP loop L13 */
  ...
  input sig1  /!*{$ belongs to L13}*/
  ...
  input sig3  /!*{$ belongs to L13}*/
```
The statement indicates the transitions containing the DMOs in the order of their appearance. Note that the \textit{belongs to} statement is attached as comment to the respective transition in the SDL specification.

A word filter for \textit{advanced ILP loops} is defined as follows:

\begin{verbatim}
/*{$ define word filter 2_to_4 }*/.
\end{verbatim}

The use of the filter is specified at places where it is required, e.g.

\begin{verbatim}
/*{$ p1 2_to_4 p2 }*/.
\end{verbatim}

5. \textbf{SEMANTIC CONSTRAINTS}

There are two possibilities to implement the ILP loop execution: a separate and an integrated implementation. In the first variant the ILP loop is executed separately; the results are stored until they are needed for protocol execution. This is a straight-forward implementation of the concept. It, however, requires an additional memory access. From the point of view of automated code generation this variant is less advantageous, because it requires separating the DMOs and the actualization of state information. Therefore, the integrated implementation is preferred, in which the ILP loop is executed together with the protocol. This, however, requires that the protocols or the respective SDL processes are in states in which they accept the respective PDUs. There may be situations in which this is not given. The ILP loop cannot be executed then. Another problem is the detection of transmission errors. They cannot be detected before executing the loop. The handling of these situations is discussed in the following.

\textit{State errors}

The execution of a PDU or a signal, respectively, may not be possible due to the current states of the SDL processes involved in the ILP loop. If a signal is executed then all processes must be in a state in which they await the PDUs contained in this signal. If these processes accept only signals exchanged within the loop and/or if they only possess one state, as supposed in [Leue96], the ILP loop can be always executed. The DMOs are processed consecutively. This is not allowed in situations when, for instance, a signal is saved by a \texttt{save} statement or when a signal is read from the input queue and discarded. Figure 4 depicts such a situation. The process \texttt{encrypt} may only encrypt video frames when it got a sending credit. Otherwise it waits in the state \texttt{wait _for_credit}. Incoming frames are saved until new credit is
available. If the ILP loop would force the encryption of incoming frames then the frames will be lost because the receiver would not accept them without credit. Besides the state would change in DMO *encryption* although it should not change at all. This can influence the protocol execution. Similar problems appear when signals are discarded in a state. In such cases the implementation has to switch to the conventional protocol execution (see below).

![Diagram](image)

**Figure 4:** Refinement of the DMOs *compress* and *encrypt*

**Transmission errors**

Transmission errors can cause execution paths other than the common path the ILP loop aims at. In a *simple ILP loop* the error is indicated after proving the checksum, because the DMOs are processed PDU-wise. The execution can immediately branch off to another execution path to handle the error. In an *advanced ILP loop* with a byte-wise processing of the signals, errors are only detected after all DMOs are completed. At that point the signals are already converted into a representation which cannot used any more for error correction, e.g. for forward error control.

**Solutions**

In order to avoid these situations appropriate measures have to be taken. Mostly a double implementation is used. Another possibility is a recovery mechanism.

*Double implementations* mean that beside the ILP implementation a conventional implementation, either a server model or a activity thread implementation, is generated. Before the processing of the PDU the states of
all protocols are checked. If a protocol is not in the appropriate state then the implementation switches to the conventional execution path. A special treatment is required for handling transmission errors when applying the **advanced ILP loop**. To ensure the correct protocol behaviour the PDU and all state information (variables, timers, states, sequence numbers) have to be stored before the ILP loop execution. When a transmission error is indicated the PDU and the original protocol state have to be restored before the conventional execution starts. Thus the signal is processed a second time to determine the kind of the error and to react appropriately.

**Recovery mechanisms** are based on log files and reset points. The actualization requires rather large efforts. Since transmission errors in fibre optics network with error rates of $10^{-15}$ are very rare the efforts for recovery mechanisms are not justified. In particular for automated code generation double implementations should be preferred, because all required information already exists and can be used by the code generator. Recovery mechanisms have to be supplemented manually in the implementation.

**Overtaking of signals**

An overtaking of signals in SDL can only take place if two or more signals are sent over the same **signalroute-channel** connection. ILP loops do not cause such a behaviour. All signals are sequentially executed within an ILP loop so that the order of the signals is preserved. However, double implementations, i.e. the combination of ILP with server model or activity thread implementations, can lead to signal overtaking. This can be avoided by applying mechanisms like the transition reordering described in [Lang99].

6. **CODE GENERATION PROCESS**

The generation of ILP implementations in COCOS is principally the same as for the other implementation techniques. It comprises two steps. First the implementation specification, i.e. the SDL specification together with the iSDL annotation, is transformed into an **intermediated representation** which consists of connected lists of the processes, of the states and of the transitions. The intermediate representation is used for various checks and optimizations [Lang99]. It is also the input for the code generation process which is carried out by the **code synthesis** component (see Figure 5). The **synthesizer** inspects the intermediate format to identify the iSDL statements which control the code generation by indicating the mapping strategy and the respective parameters. The selected implementation model identifies the respective code segments for inclusion
into the generated code sequences. The code segments contain the implementations of the SDL statements in the target language, currently in C. They may be different for certain statements, e.g. the input statement, due to the different implementation techniques. The code segments can be parametrized. The actual parameters are included during code synthesis, e.g. the signal types for each output statement. The synthesizer further accesses the code repository. It contains user implemented functions such as word filters for the advanced ILP loop as well as segments for inlining manual coded sequences. The code synthesis component also determines parameters for the initialization functions of the runtime environment, e.g. the maximum length of the common input for all SDL processes. If an ILP implementation is supposed to be generated then the ILP loop is integrated either in a server model or an activity thread implementation as shown in Figure 5.

Figure 5: Code generation in COCOS
7. MEASUREMENTS

In order to evaluate the performance of the transformation we used the protocol stack depicted in Figure 6. The protocol stack is based on the XDT (eXample Data Transfer) protocol [Koen97]. It is an example protocol used for teaching protocol engineering. XDT is a connection-oriented data transfer protocol which applies the go back $N$ error correction principle. For ILP, some data manipulation operations were added. They are located in the application layer as shown in Figure 6. Two DMOs are applied, one for calculating the checksum and one for coding/decoding the PDUs. The coding DMO applied for this experiment is very simple. It only byte-wise changes the payload data of the PDUs. The SDL/PR specification of the whole protocol stack has 991 lines. This corresponds 21 Kbyte.

![Diagram of the used protocol stack](image)

*Figure 6: Structure of the used protocol stack*

We generated a simple and an advanced ILP loop implementation and compared it with a pure server model implementation. The ILP loops comprise all transitions which belong to the checksum and coding/decoding DMOs. The loops are generated at sender and at the receiver side. The rest of the implementation is implemented straight-forward according to the server model [Lang99]. The measurement distance is the time between the sending of a PDU and the reception of the acknowledgment at sender side. It is marked by the signals start and stop in Figure 6. In each measurement cycle 10000 PDUs with a payload of 25 bytes were transferred. The measurements were repeated 50 times. Figure 7 shows the results the measurements for 3 workstations with varying performance parameters. Workstation A is a HYPER SPARC station with one processor (140 MHz) and 512 Kbyte cache, workstation B is a SPARC 20 with four processors
(125 MHz) and a cache of 256 Kbyte, whereas workstation C represents a ULTRA SPARC station with two processors and a cache of 2 Mbyte.

Figure 7: Measurement results

A comparison of the measurements at workstations A and C show that simple loop implementations bring a performance increase of about 5 to 9 per cent, while the advanced ILP loop implementations still add 7 to 11 to a total gain of almost 20 per cent. This shows that ILP may significantly improve the performance of protocol stack implementations. We further have to take the fact into account that our configuration is pretty simple. The two DMOs are adjacent and located in the same layer. Thus the cache reduces memory accesses. Several DMOs in different layers would more strongly demonstrate the effect of the approach. The phenomenon of workstation B that the server model implementation was about 1.8 per cent faster can be explained by the small cache of that workstation. The variation of the PDU execution time (measured in a separate experiment) was considerably higher at workstation B than for the other ones. This explains the fact that needed data and state information are not any more available in the cache (see also Section 2). When using larger PDUs of 200 byte payload this effect disappears (at least for low transmission error rates), because there are more memory accesses in the server model implementation (see below).

Another important aspect in this context is the influence of the error rate. Transmission errors force ILP loop results to be discarded and to start an error recovery procedure or continue with the conventional implementation. In order to determine the influence of transmission errors on the efficiency of ILP implementations we investigated the efficiency loss depending on the error rate. For this, we changed the order of the DMOs at receiver side, i.e. the checksum was calculated before the PDUs were decoded. Due to the combined execution of the operations all data are also decoded in error case. Unlike a non-ILP implementation the conventional implementation must be
re-established before error handling. We compared the modified advanced ILP loop implementation with a server model and an activity thread implementation using various error rates. This was done by falsifying a corresponding number of signals during transmission. The measurements were carried out at workstation B with a signal payload of 200 bytes and 10000 transmissions. The results are depicted in Figure 8. The measurements show that from an error rate of 6.5 or 10 per cent, respectively, upwards the activity thread and the server model implementation are more efficient. With a payload of 100 bytes this change only appears at an error rate of 13.5 or 19 per cent, respectively. The error rate is therefore not a conclusive measure for the performance of ILP implementations. The latter depends on the point where the execution stops when an error occurs and the number of operations to be repeated.

Figure 8: Influence of transmission errors on the efficiency of ILP implementations

The same experiment was also carried out with the Cadvanced code generator of the SDT tool version 3.2 [Tele98]. The execution times for error rates of 0 and 10 per cent were 65196 msec and 65342 msec, respectively, i.e. a performance gain of about 300 per cent could be achieved by using ILP compared to conventional, commercially available code generation tools.
In this paper we have presented a mapping approach for the automated derivation of ILP implementations from SDL specifications. We have discussed the semantic constraints and the code generation process. The measurements confirmed the performance increase reported from manual implementations. In addition we compared the advanced ILP loop implementation described in Section 6 with a manually coded server model implementation of the XDT protocol stack under the same conditions (200 byte payload, 10000 measurements). It showed that the manual implementation is only by a factor of 2.5 faster. Although this is only a conditional comparison, it points out that the integration of advanced manual implementation techniques like Integrated Layer Processing, activity threads or the offset technique [Lang99] may considerably improve the efficiency of automated code generation from formal descriptions.

Further research is needed to refine the techniques and to reveal places which slow down the execution speed of the generated code. One of such points is the coding/decoding of PDUs. Automated implementations of these procedures are often less efficient than manually coded ones. An inlining of manual codings in the automated code generation process might be appropriate support in this case. Moreover, it pointed out that the decision which implementation technique is selected is not straight-forward. Several experiments proved that the possible performance increase depends on the structure of the specification and on the target hardware. The better the specification supports the used implementation technique the better is the performance gain to be expected. The experiments further revealed that the different implementation techniques are not orthogonal to each other. The combination of the techniques does not bring a linear performance increase, since some optimizations are implicitly contained in other techniques. Besides the effect of certain techniques can be also achieved by combining of techniques, e.g. activity threads in combination with the offset technique can achieve similar performance parameters as ILP implementations. The selection of the appropriate techniques therefore requires a careful analysis of the specification and the implementation context. Performance predictions based on the implementation specification are a valuable means to support this process. Such a component is currently being integrated into the COCOS compiler.

An issue for a broad application of automated implementation techniques as well as other FDT based methods is the lack of complex formal specifications of real-life protocols. If larger specifications are available they are often not fully specified, especially PDU coding and decoding operations. The completion of the specification is still a time consuming process which
takes several man-months. This mostly cannot be done in an academic environment. It hinders a realistic comparison of automatically generated protocol implementations with manually coded ones in a large scale. The availability of such specifications is also decisive prerequisite for a broad and thorough application of FDT based design, validation and implementation technologies.

REFERENCES


STEPWISE DESIGN WITH MESSAGE SEQUENCE CHARTS

Ferhat Khendek¹, Stephan Bourduas¹, Daniel Vincent²

¹Department of Electrical and Computer Engineering, Concordia University
1455, de Maisonneuve W., Montréal (P.Q.), Canada H3G 1M8
E-mail: {khendek, s_bourdu}@ece.concordia.ca
²France Telecom R & D, Lannion, France

Abstract Use cases are useful in various stages of the software process. They are very often described using text that has to be interpreted by system designers. This could lead to implementation errors. Another drawback of using such informal notations is that automating the process of moving from use cases to design specification is difficult, if not impossible. It would be beneficial to represent use cases in an unambiguous way, thereby reducing the probability of misunderstanding and allowing for automation of various activities in the software process. Message Sequence Charts (MSC) is a formal language and widely used in telecommunications for the specification of the required behaviors. In this paper, we use MSC for describing use cases and we propose an approach for stepwise refinement from high-level use cases in MSC to design MSCs that contain more details about the internal components of the system and their interactions. The refinement steps are done by the designer and guided by the system architecture. For each step, the newly obtained MSC is validated automatically against the previous MSC using a conformance relation between MSCs.

1. INTRODUCTION

Distributed software systems, like any software system, go through requirement, design, implementation and testing phases. Ensuring the quality of such software systems from the initial stages is a challenging task.

UML use cases are becoming the standard form for requirements specification. An UML use case describes some functionality offered by a system as perceived by the user or an external actor of the system [1, 2, 9]. The user sees the system as a black box that responds to inputs with a specified output. Use cases are very often specified using a combination of text and diagrams that must be interpreted by the system designers and translated into a more concrete representation. It would be beneficial to represent use cases in an unambiguous way from the start, thereby reducing the probability of misunderstanding, and enabling the use of tools. MSC [3, 4, 11] is an excellent candidate as discussed in [10].

* Partially supported by France Telecom R&D.
MSC and SDL (Specification and Description Language) [5, 6] are widely used for telecommunication software engineering. In a previous work [7, 8] we have developed an approach and a tool for generating SDL specifications from a MSC and a given architecture. The given MSC is seen as a design MSC where the internal behavior of the components of the system is given. In this paper, we introduce a new approach for refining use cases specified with MSC into design MSC in a stepwise manner. The resulting MSC is then used as input for our MSC to SDL translation approach.

In our approach, a use case model is developed through interactions with the system designers and the customers as described in [1, 2, 9]. The result of this represents the functional requirements of the system under construction. Once the use cases have been agreed on, the designers must specify the architecture of the system. Guided by this architecture, the use cases are then refined in a stepwise manner. We distinguish between horizontal and vertical refinements. Horizontal refinement consists of adding new messages or actions to existing MSC axes. Vertical refinement consists of decomposing an axis into at least two other axes following the architecture of the system. The enriched MSC must conform to the previous (or parent) MSC. We therefore define a conformance relation between MSCs that is used to validate refinement steps made by the designer.

The rest of this paper is organized as follows. In Section 2, we discuss the modeling of use cases with MSC. Section 3 introduces the notion of conformance between MSCs and discusses the related work. In Section 4, Subsection 4.1 describes our stepwise refinement approach for bMSCs, while Subsection 4.2 extends this approach to HMSCs. In Section 5, we apply our approach to the ATM (Automatic Teller Machine) example, before concluding in Section 6.

2. DESCRIBING USE CASES WITH MSC

Jacobson introduced the concept of use cases in the Object-Oriented Software Engineering (OOSE) method [9]. Use cases are now part of the UML standard. A use case is an abstract description of some desired functionality provided to the user of a system. The use case description sees the system as a black box that generates some output for a given input. The fact that they are not formalized allows for flexibility in the early stages of development. This same advantage can turn into a liability if inexperienced designers are expected to interpret the use cases and then produce a design specification. In [10], the authors propose a method for formalizing use cases using MSCs. We follow this approach for the modeling of use cases.
2.1 bMSC Use Cases

bMSCs are simple diagrams that capture the interactions between system components, and between system components and the environment. For use case modeling, bMSCs are used to show the interactions between the system and the environment only as illustrated in Figure 1. The user sees a “black box” represented by a single process instance in the MSC. The axis represents the system boundary that interacts with the user. In this example, the user sends message “a” to the system, and the system reacts by sending message “b” to the user.

![Figure 1. A bMSC use case.](image)

2.2 HMSC Use Cases

HMSC allows for a more abstract view of the system [10, 11]. HMSCs improve the readability of the system by hiding low-level details and showing graphically how a set of MSCs can be combined. They are viewed as roadmaps composed of MSCs using sequential, alternative and parallel operators. HMSCs are seen as directed graphs where the nodes are: start symbol, end symbol, MSC reference, a condition, a connection point, or a parallel frame.

For simple use cases, a single bMSC is sufficient to show the behavior, but complex behaviors such as conditional executions or alternatives can lead to large and unreadable bMSCs. For these complex use cases, it is easier to use several bMSCs and then combine them using a HMSC use case.

A use case describes one possible usage of a system. Most systems will have many functions that will result in many use cases. In order to be able to organize many use cases and to allow a designer to view and navigate them with ease, a HMSC that references the HMSC use cases could be used to show an even higher level of abstraction. Using three level of abstraction allows us to specify the use cases for an entire system. Figure 2 shows how the various levels relate to each other.

The system level expresses the functional view, as a set of use cases, of the system by using a top-level HMSC. The structure level describes each use case using one HMSC without going into details. The basic level shows the interactions between the system and the environment using bMSCs. A bMSC at that level cannot refer to a HMSC as allowed in the standard [3].
bMSC at the basic level does not contain alternatives or optional behaviors. Any alternative or optional behavior can be represented using HMSC at the structure level. In this paper, we restrain the HMSC language to the weak sequencing operator, alternative operator and iteration. The parallel operator is not taken into consideration.

Figure 2. Three levels of abstraction for use case specification.

3. CONFORMANCE RELATION FOR MSC

In this section, we first introduce the definition of conformance between bMSCs, and then we extend this definition to HMSC.

3.1 bMSC

The bMSC refinement process consists of several steps. We refer by Mk to the bMSC obtained at step k. In order to preserve, throughout the refinement process, the semantics of the original use case defined by the user, each newly obtained bMSC should preserve the semantics of its parent bMSC. In other words, bMSC Mn must preserve the behavior described by bMSC Mn-1. Mn must have the events of Mn-1 and for these events Mn preserves the orders defined in Mn-1. The new messages and events introduced in Mn are not constrained. Informally, we say that a bMSC M2 preserves the behavior of (or conforms to) a bMSC M1, if and only if for each axis A1 in M1 there is a corresponding set of axes \{A21, A22, \ldots, A2n\} in M2 with all events of A1 included in \{A21, \ldots, A2n\}, and all the orders defined between events in M1 are preserved in M2.

For a formal definition of the conformance relation, we need a formal definition of a bMSC. For this, we follow the definitions in \[12, 13\].
Definition 1 (bMSC). A bMSC is a tuple \(<V, \ll, P, M, L, T, N, m>\), where
- \(V\) is a finite set of events,
- \(\ll \subseteq V \times V\) is a transitive and acyclic relation,
- \(P\) is a set of processes,
- \(M\) is a set of message names,
- \(L: V \rightarrow P\) is a mapping that associates each event with a process,
- \(T: V \rightarrow \{\text{send, receive, local}\}\) defines the types of each event as a
  send, receive or local,
- \(N: V \rightarrow M\) maps every event to a name
- \(m:\) a partial function that pairs up send and receive events.

The relation \(\ll\) is defined between the sending event \((s_m)\) and the receiving
event \((r_m)\) of message “m” as \(s_m \ll r_m\), and between events \(e_1\) and \(e_2\) in the
same axis, \(e_1 \ll e_2\), if \(e_1\) appears before \(e_2\).

Definition 2 (Conformance for bMSCs). A bMSC \(M_2 = <V_2, \ll_2, P_2, M_2,\)
\(L_2, T_2, N_2, m_2>\) conforms to a bMSC \(M_1 = <V_1, \ll_1, P_1, M_1, L_1, T_1, N_1, m_1>\),
if and only if there exist an injective mapping \(\Gamma: V_1 \rightarrow V_2\) and a surjective
function \(\phi: P_2 \rightarrow P_1\) such that:
- \(L_1(e) = \phi(L_2(\Gamma(e)))\)
- \(T_1(e) = T_2(\Gamma(e))\)
- \(N_1(e) = N_2(\Gamma(e))\)
- if \(m_1(e) = f\) then \(m_2(\Gamma(e)) = \Gamma(f)\)
- if \(e \ll_1 f\) then \(\Gamma(e) \ll_2 \Gamma(f)\)

Our conformance relation is similar to the matching relation defined in [12,
13]. However, there are two differences. The first one is related to the MSC
semantics. In fact, we do not distinguish between the visual order and the
enforced order as it is done in [12, 13]. The visual order being the transitive
and reflexive closure \(\ll^*\) of \(\ll\), while the enforced order depends on the
communication architecture where some visual orders may not hold, because
of race conditions [14]. In the formal semantics of MSC (for bMSC [15]),
the visual order has to be enforced. The question “if an architecture allows for
that order or not” is another issue. The second difference as mentioned earlier
in our conformance relation, an axis in bMSC Mn may correspond to a set of
axes in bMSC Mn+1. In other words, we allow for one-to-many relationship
between axes, contrary of the one-to-one relation in [12, 13].

Mauw and Reniers have proposed a refinement relation for interworkings
[16]. This refinement consists of decomposing an instance into its
constituents and adding internal messages between these constituents. Beside
the semantic issues, our conformance relation can be seen as a combination
of the refinement relation in [16] and the matching relation in [12, 13]. In this paper, the terms “refinement” and “conformance” are not synonymous. With the term “refinement” we only mean developing further a MSC, without automatically ensuring conformance.

For the illustration of the conformance relation, let us consider the bMSCs in Figure 3. A mapping $\Gamma$ between $M_1$ and $M_2$ that associates each event to itself and a function $\phi$ that associates each process to itself satisfy the conditions in Definition 2. bMSC $M_2$ conforms to bMSC $M_1$. We have more messages and events in $M_2$, but these events preserve the orders defined in $M_1$. The bMSC $M_3$ also conforms to $M_1$. In this case, the function $\phi$ associates $P_{21}$ and $P_{22}$ to $P_2$. In the case of bMSC $M_4$, we cannot find a mapping of events that preserves the orders defined in $M_1$. In fact, reception of message “x” by $P_{21}$ and reception of “y” by $P_{22}$ are not ordered as specified in $P_2$ in $M_1$.

![Figure 3. Examples of conformance and non-conformance between bMSCs.](image)

### 3.2 HMSC

In order to build complex behaviors from simple behaviors, the MSC standard [3] has defined weak sequential, iteration, alternative and parallel compositions of MSCs. Informally, weak composition of two bMSCs can be seen as a concatenation of the axes of the common processes in the bMSCs.

**Definition 3 (Weak composition of bMSCs).** Given two bMSCs, $M_1 = \langle V_1, \langle\langle_1, P_1, M_1, L_1, T_1, N_1, m_1\rangle \rangle$ and $M_2 = \langle V_2, \langle\langle_2, P_2, M_2, L_2, T_2, N_2, m_2\rangle \rangle$, with $V_1 \cap V_2 = \emptyset$, $M_1 \circ M_2 = \langle V_1 \cup V_2, \langle\langle, P_1 \cup P_2, M_1 \cup M_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2\rangle \rangle$, where $\langle\langle = \langle\langle_1 \cup \langle\langle_2 \cup \{(e_1, e_2), \text{ such that } L_1(e_1) = L_2(e_2), e_1 \in V_1 \text{ and } e_2 \in V_2\}$.}
From the syntactical point of view, a HMSC defines a roadmap where MSCs are combined using weak sequential, iteration, alternative and parallel compositions. As mentioned in Section 2, we do not take into account the parallel operator in this paper. From a semantic point of view, we define a HMSC as a potentially infinite set of alternatives of (infinite) weak sequential composition of bMSCs.

**Definition 4 (HMSC).** A HMSC $H_1$ is a set of sequences $seq_i$ (or bMSCs), with $seq_i = M_{i1}oM_{i2}M_{i3}o...oM_{iq}$, where $M_{ij}$ is a bMSC, for $j = 1, \ldots, q$ and $i = 1, \ldots, n$.

The set and the sequences could be infinite. The iteration is defined as the repetition, using weak sequential composition, of the same bMSC.

Informally, we say that a HMSC $H_1$ conforms to HMSC $H_2$, if and only if for each alternative bMSC $M_1$ in $H_1$ there is a bMSC $M_2$ in $H_2$, such that $M_2$ conforms to $M_1$. $M_2$ preserves the behavior of $M_1$. $H_2$ may contain more alternatives than $H_1$, but each alternative of $H_1$ is preserved in at least one alternative of $H_2$.

**Definition 5 (Conformance for HMSCs).** Given two HMSCs, $H_1 = \{seq_{1i}, i = 1, \ldots, n\}$ and $H_2 = \{seq_{2j}, j = 1, \ldots, m\}$, $H_2$ conforms to $H_1$, if and only if for each $seq_{1i}$ in $H_1$ there exist $seq_{2j}$ in $H_2$ such that $seq_{2j}$ conforms to $seq_{1i}$.

The conformance between $seq_{1i}$ and $seq_{2j}$ is given in Definition 2. Both conformance relations (for bMSCs and HMSCs) are transitive and reflexive.

### 4. FROM USE CASES TO DESIGN SPECIFICATION

Typically, designers will use their experience to generate design specifications from textually represented use cases. This method can lead to errors in implementation if the use cases are unclear. How these types of errors can be avoided is the goal of our approach. It enables a designer to specify an initial use case in MSC and refine it incrementally into a MSC design specification. The system architecture plays an important role in the stepwise refinement of MSC use cases. A refined MSC must conform to its parent MSC and follow the underlying system architecture. This system architecture can be represented using either UML or SDL. Our methodology for stepwise refinement of use cases is to be used in conjunction with existing tools for translation of a UML architecture into a SDL architecture [17], and the generation of a SDL specification from a given target architecture and MSCs [8].
4.1 bMSC Refinement Methodology

During the refinement process, we distinguish between vertical and horizontal refinements. Vertical refinement reflects the architectural decisions, while horizontal refinement allows to enrich the behavior of a bMSC. Both types of refinements are used together to incrementally refine a use case bMSC into a design specification. Figure 4 illustrates how both types of refinements are combined to reach a design specification.

The refinement process is dependent on the system architecture. The amount of layers used in the architecture limit the number of vertical refinements. There can only be one vertical refinement per layer. Each vertical refinement step splits a bMSC instance into several instances according to the architecture. The instances that are not decomposed keep the same names. When an instance is decomposed the designer has to distribute its events among the new instances. For each message “m”, the names of the associated events (sending and receiving) are kept unchanged. The messages and events associated with instances that are not refined are kept unchanged and re-generated automatically. In general, a combination of vertical and at least one horizontal refinement is needed in order to generate a MSC that may conform to the previous level.

Horizontal refinement is concerned with adding messages, events and local actions to the bMSC. The designer can add new messages, sending and receiving events, as well as local actions. The messages that can be used in a horizontal refinement are specified in the architecture. A designer cannot introduce messages that are not specified in the architecture at the current level of refinement. Unlike vertical refinement, there is no limit to the number of horizontal refinements that can be performed at any given layer.
After each horizontal refinement, the designer can check automatically whether the refined bMSC conforms to the previous one. This verification of the conformance relation between bMSCs is implemented with the Event Order Tables (EOT) introduced in [7, 8]. As mentioned earlier, the bMSCs used in this paper do not contain alternative or optional behaviors.

**Event Order Tables (EOT)**

An EOT for a bMSC is a matrix that shows precedence relationships between events of the MSC. Figure 5 shows a bMSC and its corresponding EOT. Each message between two instances corresponds to two events: sending and receiving. Once the events on a MSC have been labeled appropriately, the EOT can be constructed. Following the MSC semantics and the relation $\ll$, we use two rules to generate the EOT [8]:

- Each instance is totally ordered (except for co-regions where there is no order between events).
- A reception event happens after its matching sending event.

Inspection of each process axis in Figure 5 reveals that the first rule must be applied to P2, yielding the relation $e_2 \ll e_3$. Applying the second rule to the MSC of Figure 5 yields \{e_1 \ll e_2, e_3 \ll e_4\}. The transitive closure $\ll^*$ of $\ll$ allows for the construction of the EOT. For instance, $e_1 \ll e_2$ and $e_2 \ll e_3$ implies $e_1 \ll e_3$. A cell in the EOT is set to true if the row event occurs before the column event.

**Definition 6 (Inclusion for EOTs).** We say that $EOT T_1$ is included in $EOT T_2$, if and only if

- all the events of $T_1$ are in $T_2$, and
- for each pair of events $(e_i, e_j)$ if $e_i \ll e_j$ in $T_1$ then $e_i \ll e_j$ in $T_2$.

Provided the relation, enforced by the bMSC refinement approach (and tool), between instances and events in bMSC $M_{k+1}$ and bMSC $M_k$, we say that $M_{k+1}$ conforms to $M_k$ if and only if EOT of $M_k$ is included in EOT of $M_{k+1}$.

**Proposition 1 (Conformance of bMSCs using EOTs).** Given bMSCs, $M_k$ and $M_{k+1}$, provided that events in $M_k$ are mapped into themselves in $M_{k+1}$, $M_{k+1}$ conforms to $M_k$ if and only if EOT of $M_k$ is included in the EOT of $M_{k+1}$.
**Proof.** The proof is straightforward. Our refinement approach maps events in \( M_k \) into themselves in \( M_{k+1} \). The orders defined in \( M_k \) are preserved in \( M_{k+1} \) if and only if the EOT of \( M_k \) is included into the EOT of \( M_{k+1} \). [End]

Figure 6 shows a bMSC that conforms to the bMSC in Figure 5, the message “c” sent from P2 to P3 has been added. In order to preserve the original event labels, the event labels \( e_5 \) and \( e_6 \) have been used for the new events.

![Refined bMSC](image)

*Figure 6. Example of conformance verification using EOTs.*

The EOT in Figure 5 specifies an ordering of events that is maintained by the refined bMSC. In fact, the EOT in Figure 5 is included in the EOT in Figure 6. The old relations are bolded in Figure 6; as long as these are not violated, the new bMSC conforms to its parent bMSC.

### 4.2 Stepwise Refinement of HMSCs

A HMSC is composed of a number of bMSCs following a certain roadmap. To enrich a use case HMSC, we keep the same roadmap and we refine the bMSCs. A HMSC is refined according to the following rules:

1. Keep the roadmap unchanged.
2. For vertical refinement, all bMSCs are vertically refined at the same time so that each bMSC has the same number of instances.
3. For horizontal refinement, we refine horizontally the referenced bMSCs. Each bMSC can be refined horizontally independently of the others.

Vertical refinements are done automatically according to the system architecture and the current level of abstraction. The user refines bMSCs horizontally using the method described previously and messages described
in the architecture at the current level of abstraction. Unfortunately, the conformance of the refined bMSCs to their corresponding bMSCs does not lead automatically into the conformance of the refined HMSC to the original HMSC. In fact, orders between events in a given axis may not hold anymore when the axis is decomposed and the events distributed among the refining axes as shown in Figure 7. Indeed, in the HMSC M1 o M2, the sending of “x” always precedes the sending of “y”, but this not the case in the HMSC M1’ o M2’, because of the distribution of these two events. In this example, M1’ conforms to M1 and M2’ conforms to M2, but M1’ o M2’ does not conform to M1 o M2.

![Figure 7. Non-conformance between HMSCs.](image)

**Theorem 1 (Conformance between HMSCs).**
Given a HMSC H1 and its refinement H2, (with the same roadmap), if
- each bMSC in H2 conforms to its corresponding bMSC in H1, and
- in the HMSC H2, for each pair of bMSCs, M1’ and M2, such as M1’ o M2’ in H2, for each set of axes \{A_{11}, \ldots, A_{1n}\} in M1’ and M2’ resulting from the decomposition of A1 (in M1 and M2 in H1), the order between the last event of A1 in M1 and the first event of A1 in M2 is preserved,

then H2 conforms to H1.

**Proof.** Consider two HSMCs, H1 and its refinement H2 obtained following the rules mentioned above. From a semantic point of view, each HMSC is a set of alternatives of concatenation of bMSCs. Roadmaps of H1 and H2 are identical, each bMSC in H1 has a corresponding bMSC in H2. Any sequence of concatenation seq1 in H1 has its corresponding sequence of concatenation seq2 in H2, using corresponding bMSCs in H2. In order to show that H1 conforms to H2, we have to show that seq2 conforms to seq1. For that, we only have to prove that if M1’ conforms to M1, M2’ conforms to M2, and the second condition of the theorem holds, M1’ o M2’ conforms to M1 o M2.
M1’ conforms to M1 means that orders in M1 are preserved in M1’. M2’ conforms to M2 also means that orders of M2 are preserved in M2’. M1 o M2 (before decomposition of axes) imposes orders on events in the same axes and these orders have to be preserved in M1’ o M2’. The second condition of the theorem ensures the preservation of these orders in M1’ o M2’. Therefore, M1’ o M2’ conforms to M1 o M2.

During the refinement of HMSC use cases, we check the conformance between the new HMSC and its parent HMSC by checking the conformance between each pair bMSCs and the second condition of the theorem above. When a bMSC in the new HMSC does not conform to its corresponding bMSC in the parent HMSC, the refinement is not accepted. New refinements and modifications have to be applied to the new bMSC and the conformance checked again. In some cases, the designer has to add new messages in the refined HMSC to ensure that the second condition of the theorem holds. The tool can also add automatically “dummy” messages to satisfy this condition of the theorem. Notice that in our refinement approach the conformance of bMSCs is a sufficient condition and not a necessary one. However, in order to ensure the conformance between HSMCs, we enforce conformance between pairs of bMSCs. The study of all the possible necessary conditions is left for future investigations.

5. APPLICATION

Automation (tool) reduces the likelihood of human error and increases the confidence in the generated specifications.

5.1 Tool

We have developed a MSC refinement tool. The use cases are specified with HMSC and the system architecture is specified using UML (or SDL directly). The stepwise refinement of MSCs is used in conjunction with other methodologies that have been automated, namely:

- UML to SDL architectural translation [17].
- Generation of SDL specification from a given SDL architecture and MSC [7, 8]. The SDL specification preserves the behaviors specified in the MSC and is free of any design error such as deadlocks.

Figure 8 shows how the MSC refinement tool can be combined with existing tools. The shaded region represents the MSC refinement process. A tool guides the user through the refinement process and verifies that the resulting MSCs conform to the original use cases and follow the system architecture.
The MSC refinement approach (and tool) can also be used in the front-end of the SDL specification enrichment approach introduced in [18].

5.2 Example

This section illustrates our refinement approach through an example. Figure 9 shows a system architecture representing an ATM machine using SDL. There are two processes, namely “ATM” and “BANK”. There is a channel that allows the user of the system to interact with the ATM process instance, and another channel allows for the two process instances to interact with each other. The signals these processes exchange with each other and their environment are specified in the architecture.

Two functions provided by an ATM machine are the ability to withdraw and deposit money. Both these functions have a common starting point. The user must first insert a card and enter the correct PIN before a transaction can occur. Figure 10 shows the system and structure level HSMCs.
Figure 10. System and structure level HMSCs for the ATM example.

The system level HMSC shows three different use cases, “Exit”, “Deposit” and “Withdraw”. It also shows that each use case has the common behavior denoted by the “Login” block. The structure level shows more details. The “Login” HMSC has only one bMSC, “Valid_Pin”. Later on, the system designer can decide to add an “Invalid_Pin” scenario to the “Login” HMSC that would not affect the rest of the system. The “Withdraw” HMSC shows that the withdraw use case can have two results, “WithdrawOK” and “WithdrawNOK”. For illustration purpose, we will perform refinements on “WithDrawOK” only. The bMSC “WithdrawOK” in Figure 11.a is first refined vertically into the bMSC in Figure 11.b. Note that there is no new behavior added, only a splitting of the instance “ATM_Machine” to reflect the architecture of Figure 9.

Figure 11. Vertical and horizontal refinements of “WithdrawOK”.

Once the vertical refinement has been performed, the designer can now enrich the bMSC. A horizontal refinement is shown in Figure 11.c. The new MSC conforms to the original MSC in Figure 11.a.
Now suppose that the ATM block of Figure 9 was further decomposed into three separate processes: Dialogue, MoneyHandler and Printer. With our refinement process, the MSC of Figure 11.c is first refined vertically to show the decomposition of the ATM block. This refinement is followed by a horizontal refinement with the addition of messages. The resulting MSC, that conforms to the MSC in Figure 11.c and therefore to the initial use case in Figure 11.a, is shown in Figure 12.

![Figure 12. Another refinement of “WithdrawOK”](image)

While horizontal refinements are being done separately for each bMSC of the HMSC. The designer can verify the conformance of each new bMSC (HMSC) against its corresponding bMSC (HMSC) in the previous stage.

6. CONCLUSION

Use cases are a good way to capture user requirements. They are abstract and are often described using textual documents that can be misinterpreted later on by designer. The reason for this is that designers apply their experience to translate use cases into a design and implementation. A structured way of going from use cases to specification would benefit the software development process by detecting and avoiding errors early on, and by enabling easier software comprehension. This will reduce development and maintenance costs. MSCs are well suited for this because they can show a system at a high level of abstraction, or they can be used for low-level specifications. They are intuitive, easy to learn and formally defined.

The goal of this project is to create a methodology and a CASE tool to semi-automate the process of refining a use case HMSC into a design specification. The MSC refinement tool is used as a front-end tool for an existing set of tools for translating UML architecture specifications to SDL architecture
specifications, and from MSC specifications to SDL specifications. This set of tools aims at improving the quality of the software as well as shortening the development process.

Other issues such as conformance relations that allow for enrichment and modification of roadmaps are under consideration.

**Acknowledgements:** We would like to thank Loïc Helouët from France Telecom for his very helpful comments.

**REFERENCES**

FORMAL SYNTHESIS AND CONTROL OF SOFT EMBEDDED REAL-TIME SYSTEMS

Pao-Ann Hsiung
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi–621, Taiwan, ROC
hpa@computer.org

Abstract
Due to rapidly increasing system complexity, ever-shortening time-to-market, and growing demands for soft real-time, formal methods are becoming indispensable in the synthesis of embedded real-time systems. In this work, a formal method based on Time Free-Choice Petri Nets (TFCPN) is proposed for synthesizing and controlling Soft Embedded Real-Time Systems (SERTS). Technically, the proposed method employs quasi-static data scheduling for satisfying limited embedded memory requirements and controls firing interval bounds for satisfying soft real-time constraints. An application example is given to illustrate the feasibility of the formal method, which can also be used for code generation.

Keywords: embedded real-time systems, formal synthesis, control, quasi-static scheduling, Time Free-Choice Petri nets

1. INTRODUCTION
With the proliferation of Soft Embedded Real-Time Systems (SERTS) into our daily lives in the form of home appliances, internet appliances, personal assistants, wearable computers, telecommunication gadgets, and transportation facilities, we are now faced with a growing escalation of system complexity, ever-shortening time-to-market, and growing demands for soft real-time applications. All these factors have propelled the need for practical formal methods that can be used to synthesize and control SERTS. A large portion of formal methods is mainly devoted to hard real-time system analysis. In contrast, we show how formal methods can also be applied to soft real-time systems that have flexible ranges of acceptable behaviors.

Our target systems are soft real-time systems such as multimedia servers, communication networks, telecommunication devices, home electric appliances, and information appliances. These systems can tolerate deadline misses up to a certain threshold value. Not every deadline needs to be met, only most of the deadlines need be met. Further, deadlines themselves can be specified as
a time interval \((\alpha, \beta)\) such that if a system task completes execution no earlier than \(\alpha\) and no later than \(\beta\), then the task does not miss its deadline, where \(\alpha\) and \(\beta\) are integers representing some points in the time-line.

Informally, our target problem is to synthesize an embedded real-time system starting from an initial set of loose specifications into a final set of strict specifications such that the final specification satisfies all user-given real-time constraints such as response times, deadline, and periods. A specification \(\psi_1\) is said to be looser than another specification \(\psi_2\) if all the behaviors given by \(\psi_2\) are implied by \(\psi_1\). In plain terms, our solution is to restrict loose specifications into stricter ones such that given constraints are met.

The two main issues involved in the design of SERTS are as follows:

- **Bounded Memory Execution**: A processor cannot have infinite amount of memory space for the execution of any software process. This fact is even more emphasized in an embedded system, which generally has only a few hundreds of kilobytes memory installed. Thus, a SERTS must utilize as less memory as possible.

- **Soft Real-Time Constraints**: A processor may have to execute several concurrent tasks with precedence and temporal constraints. Thus, a SERTS is generally composed of several soft concurrent real-time tasks.

Our formal model is based on the recently proposed *Time Free-Choice Petri Nets* (TFCPN) [15], which is a sub-class of Time Petri Nets. In solution to the above two issues, our proposed method consists of the following two phases:

- **Quasi-Static Data Scheduling** (QSDS): This scheduling phase ensures that embedded real-time applications do not require an unbounded amount of memory for execution, since embedded real-time systems have limited amount of embedded memory,

- **Firing-Interval Bound Synthesis** (FIBS): This synthesis phase ensures that an embedded real-time system meets all soft real-time constraints, which are generally modeled as action or firing time intervals. This phase is also called *Controller Synthesis* since controllers can be synthesized for soft real-time systems using this method.

Software code can also be generated for soft real-time systems by applying our synthesis method to a recently proposed code generation scheme, which was for hard real-time systems [15]. Due to page-limits, we will not delve into this part of the work in this article.

This article is organized as follows. Section 2 gives some previous work related to SERTS synthesis. Section 3 will formulate, model, and solve the SERTS synthesis problem. Section 4 will illustrate the proposed method through an application example. Section 5 will conclude the article with some research directions for future work.
2. PREVIOUS WORK

Currently, synthesis of soft real-time systems is a hot topic of research in the field of hardware-software codesign of embedded systems [11]. Previously, a large effort was directed towards synthesis of hard real-time systems, especially in the application of formal methods. Synthesis was mainly carried out for communication protocols [19], plant controllers [4, 18, 5], and real-time schedulers [25, 1] because they generally exhibited regular behaviors. Only recently has there been some work on automatically generating code for embedded systems [17, 16, 23, 26, 6]. In the following, we will briefly survey the existing works on the synthesis of non real-time software and controller synthesis, on which our work is based.

Lin [16, 17] proposed an algorithm that generates a software program from a concurrent process specification through intermediate Petri-Net representation. This approach is based on the assumption that the Petri-Nets are safe, i.e., buffers can store at most one data unit, which implies that it is always schedulable. The proposed method applies quasi-static scheduling to a set of safe Petri-Nets to produce a set of corresponding state machines, which are then mapped syntactically to the final software code. Later, Zhu and Lin [26] proposed a compositional version of the synthesis method that reduced the generated code size and was thus more efficient.

A software synthesis method was proposed for a more general Petri-Net framework by Sgroi et al. [23]. A quasi-static scheduling algorithm was proposed for Free-Choice Petri Nets (FCPN) [23]. A necessary and sufficient condition was given for a FCPN to be schedulable. Schedulability was first tested for a FCPN and then a valid schedule generated by decomposing a FCPN into a set of Conflict-Free (CF) components which were then individually and statically scheduled. Code was finally generated from the valid schedule.

Balarin et al. [6] proposed a software synthesis procedure for reactive embedded systems in the Codesign Finite State Machine (CFSM) [7] framework with the POLIS hardware-software codesign tool [7]. This work cannot be easily extended to other more general frameworks.

Besides synthesis, there are also some recent work on the verification of software in an embedded system such as the Schedule-Verify-Map method [12], linear hybrid automata techniques [10, 13], and mapping strategy [8].

Controller synthesis for plants (also called supervisor synthesis) was mainly performed in the discrete time domain, with a large portion of classical work done by Ramadge and Wonham [21, 22]. Around 1994, when timed automata was proposed as a dense-time model for real-time systems [3], controller synthesis was extended to dense real-time systems [4, 18, 25] as well as to hybrid systems [24]. Recently, the same technique was further extended to multimedia scheduler synthesis [1]. Given a dense real-time system modeled by timed
automata and a (temporal) property given as a formula in *Timed Computation Tree Logic* (TCTL) [2, 9], a controller is synthesized such that it restricts the behavior of the system for satisfying the property. This is the controller synthesis problem. Recently, system parameters have also been taken into consideration for real-time controller synthesis [14].

3. FORMAL SYNTHESIS AND CONTROL

A formal synthesis method for soft embedded real-time systems is presented in this section. Its basic features are that the synthesized system executes in bounded memory and satisfies all user-given soft real-time constraints. Before going into details, the system model and related terminologies are presented.

A soft embedded real-time system is specified as a set of *Time Free-Choice Petri Nets* (TFCPN), which are time extensions of Free-Choice Petri Nets (FCPN) [23]. As mentioned in Section 2, FCPN was used for the quasi-static scheduling of embedded real-time software. But, there was no concept of time in the FCPN model, which makes it an unconvincing model for real-time systems. FCPN was recently extended to include time just as Merlin and Farber’s *Time Petri Nets* (TPN) [20] are time extension of standard Petri Nets. The extended version called TFCPN was introduced in [15] and is presented here.

In the rest of this section, we first define TFCPN, give its properties, and explain why TFCPN are used for modeling SERTS. Then, the problem to be solved is formulated. Finally, our proposed synthesis algorithm is described.

3.1 System Model

**Definition 1:** *Time Free-Choice Petri Nets* (TFCPN)

A *Time Free-Choice Petri Net* is a 5-tuple \((P, T, F, M_0, \tau)\), where: \(P\) is a finite set of places, \(T\) is a finite set of transitions, \(P \cup T \neq \emptyset\), and \(P \cap T = \emptyset\), \(F : (P \times T) \cup (T \times P) \rightarrow N\) is a weighted flow relation between places and transitions, represented by arcs, such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition (this is called
Free-Choice), where $N$ is a set of nonnegative integers, $M_0 : P \rightarrow N$ is the initial marking (assignment of tokens to places), and $\tau : T \rightarrow Q^* \times (Q^* \cup \infty)$, i.e., $\tau(t) = (\alpha, \beta)$, where $t \in T, \alpha$ is the earliest firing time (EFT), and $\beta$ is latest firing time (LFT); together they are called firing interval bounds (FIB).

Graphically, a TFCPN can be depicted as in Fig. 1, where circles represent places, vertical bars represent transitions, arrows represent arcs, black dots represent tokens, and integers labeled over arcs represent the weights as defined by $F$. Here, $F(x, y) > 0$ implies there is an arc from $x$ to $y$ with a weight of $F(x, y)$, where $x$ and $y$ can be a place or a transition. Conflicts are allowed in a TFCPN, where a conflict occurs when there is a token in a place with more than one outgoing arc such that only one enabled transition can fire, thus consuming the token and disabling all other transitions. For example, $t_2$ and $t_3$ are conflicting transitions in Fig. 1. But, confusions are not allowed in TFCPN, where a confusion is a result of coexistence of concurrency and conflict. An example of confusion is given in Fig. 2. Transitions $t_1$ and $t_2$ are concurrent and $t_3$ and $t_4$ are in conflict.

By disallowing confusions, a system modeled by TFCPN can be easily analyzed and synthesized because conflicts can be resolved through net decomposition of a TFCPN into conflict-free components. Though the free-choice restriction disables a designer from describing systems that have coexisting concurrency and conflicts (i.e. synchronization with conflict as in Fig. 2), yet the net decomposition approach can be extended to general Petri nets, which will be part of our future research.

Semantically, the behavior of a TFCPN is given by a sequence of markings, where a marking is an assignment of tokens to places. Formally, a marking is a vector $M = \langle m_1, m_2, \ldots, m_{|P|} \rangle$, where $m_i$ is the non-negative number of tokens in place $p_i \in P$. Starting from an initial marking $M_0$, a TFCPN may transit to another marking through the firing of an enabled transition and re-assignment of tokens. A transition is said to be enabled when all its input places have the required number of tokens for the required amount of time, where the required number of tokens is the weight as defined by the flow relation $F$ and the required amount of time is the earliest starting time $\alpha$ as defined by $\tau$. An enabled transition upon firing, the required number of tokens are removed from all the input places and the specified number of tokens are placed in the output places, where the specified number of tokens is that specified by the flow relation $F$ on the connecting arcs. An enabled transition may not fire later than the latest firing time $\beta$ defined by $\tau$.

SERTS have both data-dependent executions as well as time-dependent specifications. Both of these characteristics are well-captured by TFCPN. TFCPN can distinguish clearly between choice and concurrency, hence they are good models of data-dependent and concurrent computations. Further, TFCPN can
also distinguish clearly between data-dependent and time-dependent choices, thus TFPCN are well-defined models for our target SERTS.

Some properties of Petri Nets (PN) can be defined as follows. \textit{Reachability}: a marking $M'$ is reachable from a marking $M$ if there exists a firing sequence $\sigma$ starting at marking $M$ and finishing at $M'$. \textit{Boundedness}: a PN is said to be $k$-bounded if the number of tokens in every place of a reachable marking does not exceed a finite number $k$. A safe PN is one that is 1-bounded. \textit{Deadlock-free}: a PN is deadlock-free if there is at least one enabled transition in every reachable marking. \textit{Liveness}: a PN is live if for every reachable marking and every transition $t$ it is possible to reach a marking that enables $t$.

### 3.2 Problem Formulation

In multimedia presentations, network computing, distance learning, and other soft real-time systems, the real-time behavior can be \textit{controlled}, that is, restricted such that the system satisfies some pre-defined specification. For example, if the tolerable network lag in some kind of network computing is pre-specified as 10 seconds, then the behavior of the network computing environment could be controlled such that under all circumstances a maximum of 10 seconds network lag is encountered during computation.

To model the above soft real-time behavior, we define a new simplified linear temporal logic, which a controller is supposed to enforce in a SERTS.

**Definition 2 : Timed Reachability Specification**

A \textit{Timed Reachability Specification} (TRS) for a TFPCN $A = (P, T, F, M_0, \tau)$ has the following syntax:

$$\phi ::= \exists \tau \ast c \overrightarrow{\phi} \mid \forall \boxdot \ast c \overrightarrow{\phi} \mid \phi_1 \land \phi_2$$  \hspace{1cm} (1)

where $\ast \in \{<, \leq, =, \geq, >\}$, $\overrightarrow{\phi}$ is a non-negative integer vector of $|P|$ elements, and $\phi_1$ and $\phi_2$ are TRS formulae.

Semantically, $\exists \tau \ast c \overrightarrow{\phi}$ means eventually and obeying the timing restriction $\ast c$ there exists a TFPCN marking $M$ such that $M = \overrightarrow{\phi}$, where $\overrightarrow{\phi}$ is a \textit{token assignment} represented by a non-negative integer vector of $|P|$ elements such that each element represents the amount of tokens that must reside in the corresponding place. This definition is the same as a marking, but we do not call it a marking because $\overrightarrow{\phi}$ might not be reachable from the initial marking. Further, $\forall \boxdot \ast c \overrightarrow{\phi}$ means for all reachable markings $M$, while obeying the timing restriction $\ast c$, $M = \overrightarrow{\phi}$. Thus, a TRS gives a linear temporal condition that a TFPCN must satisfy. Since we consider a single microprocessor (executing software) in our soft embedded real-time systems, linear temporal logic in the above TRS form (Equation 1) is sufficient for expressing all reachability properties such as safeness, deadlines, boundedness, deadlock-free, and starvation.
### Table 1. Soft Embedded Real-Time System Synthesis Algorithm

\[
\text{SERTS\_Synthesize}(S, \mu, \phi)
\]

Set of TFCPN \( S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n \} \);

Integer \( \mu \); // Maximum memory

TRS \( \phi \); // Specification

\{

// Quasi-Static Data Scheduling (QSDS)

for each \( A_i \) in \( S \) { (1)

\( B_i = \text{CF\_Generate}(A_i) \); // \( B_i \): set of CF components (2)

for each CF component \( A_{ij} \) in \( B_i \) { (3)

\( QSS_{ij} = \text{Quasi\_Static\_Schedule}(A_{ij}, \mu) \); // \( QSS \): schedules (4)

if \( QSS_{ij} = \text{NULL} \) { (5)

print “QSDS failed for \( A_{ij} \)”;

return \text{QSDS\_Error}; (6)

} else \( QSS_i = QSS_i \cup \{QSS_{ij}\} \) } (7)

// Firing Interval Bound Synthesis (FIBS)

if \( \text{Controller\_Synthesize}(S, QSS_1, \ldots, QSS_n, \phi) = \text{NULL} \)

return \text{FIBS\_Error}; (9)

else return \text{Synthesized}; (10)

\}

Other properties which are not as important for SERTS such as liveness cannot be specified using TRS.

Given a system model TFCPN (Definition 1) and a specification logic (Definition 2), we are now ready to formulate our problem as follows.

**Definition 3 : Soft Embedded Real-Time System Synthesis**

Given a system modeled by a set of TFCPN \( S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n \} \) and a specification \( \phi \) in TRS, the system description \( S \) is to be synthesized by scheduling and by modifying firing interval bounds such that \( S \) is made to satisfy \( \phi \).

### 3.3 Synthesis Algorithm

As introduced in Section 1 and formulated in Definition 3, there are two objectives for our SERTS synthesis algorithm, namely bounded memory execution and soft real-time constraints satisfaction. Thus, the algorithm \text{SERTS\_Synthesize()} proposed in Table 1 is intuitively divided into two phases corresponding to the two objectives.
As shown in Table 1, given a set of TFCPNs \( S = \{ A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n \} \), a maximum bound on memory \( \mu \), and a TRS \( \phi \), a system is synthesized upon completion of the following two phases.

### 3.3.1 Quasi-Static Data Scheduling (QSDS)

The basic concept here is to employ net decomposition such that firing choices that exist in a TFCPN are segregated into individual Conflict-Free (CF) components. This is done by a procedure \texttt{CF\_Generate()} as in Step (2) for each \( A_i \), which results in a set \( B_i \) of CF components corresponding to \( A_i \). The CF components are not distinct decompositions as a transition may occur in more than one component. As in Step (4), each CF component of each TFCPN is quasi-static scheduled, that is, starting from an initial marking for each component, a finite complete cycle is constructed, where a finite complete cycle is a sequence of transition firings that returns the net to its initial marking. A CF component is said to be schedulable if a finite complete cycle can be found for it and if it is deadlock-free. Once all CF components of a TFCPN are scheduled, a valid quasi-static data schedule \( QSS_i \) for the TFCPN \( A_i \) can be generated as a set of the finite complete cycles. The reason why this set is a valid schedule is that since each component always returns to its initial marking, no tokens can get collected at any place. Details of this procedure can be found in [23].

We have extended the quasi-static scheduling approach given in [23] to consider timing constraints on transition firings during the scheduling process. A quasi-static schedule is said to be feasible only if all transition firing intervals are satisfied. Satisfaction of memory bound can be checked by observing if the memory space represented by the maximum number of tokens in any marking does not exceed the bound. Here, each token represents some amount of buffer space (i.e., memory) required after a computation (transition firing). Hence, the total amount of actual memory required is the memory space represented by the maximum number of tokens that can get collected in any marking, which results from the transition firings in a quasi-static data schedule.

### 3.3.2 Firing Interval Bound Synthesis (FIBS)

This phase consists of a procedure \texttt{Controller\_Synthesize()} as in Step (9) of Table 1, which synthesizes a controller for system \( S \) with quasi-static schedules \( QSS_1, \ldots, QSS_n \) to satisfy a TRS \( \phi \).

Some embedded soft real-time systems, such as multimedia and networks, can tolerate latencies that occur due to network lags, inferior display technologies, weak processing power, and limited memory bandwidth. In order to control such systems, normally a controller is needed to ensure quality of service (QOS), predictability, and reliability. The two main issues involved in the design of a controller for embedded soft real-time systems are as follows:
- **Synchronization Wait**: A software task, upon completion of its scheduled jobs, may have to wait for a period of time to synchronize with another software task or with the hardware.

- **Real-Time Specification**: In order to satisfy some given real-time specification, such as deadlines, a software task must finish execution of its scheduled jobs earlier than system-permitted deadlines.

Solving the above two issues, a synthesis method must generate a controller that ensures all synchronizations and real-time specifications are met. In our proposed method, the above two issues are solved as follows. Here, each software task $T$ is associated with a time interval $(\alpha, \beta)$, where $\alpha$ is the earliest start time of $T$ and $\beta$ is the latest finish time of $T$.

- **Postpone Release Time**: For synchronization to be feasible and for predictable behavior, a software task that needs to wait for some other tasks, should have its earliest start time $\alpha$ changed into $\alpha + \delta_w$, where $\delta_w > 0$ is the amount of wait time required.

- **Advance Finish Time**: For satisfaction of real-time specifications, the deadline of a software task is advanced from $\beta$ to $\beta - \delta_h$, where $\delta_h > 0$ is the difference in the user-specified and system-permitted deadlines.

As shown in Table 2, a solution to FIBS is proposed as an algorithm **Controller Synthesize**, which consists of three nested for-loops spanning over each TFCPN (Step (1)), over each schedule of a TFCPN (Step (2)), and over each transition in a schedule (Step (3)). **Firing Interval Bound Synthesis** or **Controller Synthesis** mainly restricts some transition firing interval $\tau(t) = (\alpha, \beta)$ into a smaller interval $(\alpha', \beta')$, where $\alpha' \geq \alpha$ and $\beta' \leq \beta$, such that a given TRS formula is satisfied. In the above case, $(\alpha, \beta)$ is said to be **less restricted** than $(\alpha', \beta')$.

The conditions given in Step (3) of Table 2 specifies that we consider only each prefix $t^i = \langle t_0, t_1, \ldots, t_k \rangle$ of a schedule $v_{ij}$ that leads to a possible token assignment specified in some component of $\phi$. A transition in $\text{in\_trans}(p)$ is an incoming transition of place $p$ and the function $\text{token}_{\phi_i}(p)$ gives the number of tokens at place $p$ specified in the $i$th component $\phi_i$ of $\phi$. First, as in Step (4) the aggregate delay interval $\tau$ is calculated for a schedule prefix $t^i$ by summing up all the EFT $\alpha_i$ and all the LFT $\beta_i$ of transitions $t_i$ in $t^i$. Then, a full set of new interval bounds (New_IBS$_i$) is constructed by procedure **IBS_Synthesize()** in Step (5), with details in Table 3.

Corresponding to the two kinds of path-formulae in a TRS $\phi$, there are two ways for incorporating the new set of interval bounds in $S$.

1. $\phi = \exists \circ_{c_i} p_i^g$: A variable Min_IBS$_i$ keeps track of the set of minimally restricted transition firing intervals of $A_i$ for satisfying $\phi$ (Steps (6) and (7)). A solution
Controller_Synthesize \((S, QS S_1, \ldots, QS S_n, \phi)\) 

set of TFCPN \(S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}\); 

TRS \(\phi = M_1 \phi_1^* \land M_2 \phi_2^* \land \ldots, M_n \phi_n^*, \) where \(\phi_i = M_i \phi_i^*, M_i \in \{\exists \square \land, \forall \square \land\}, \phi_i^* = (x_1, x_2, \ldots, x_{|P_i|}), x_i \in N_{\geq 0}; \) 

\[
\text{for } i = 1, \ldots, n \{ \text{(1)} \\
\quad \text{for each schedule } v_{ij} \in QS S_i \{ \text{(2)} \\
\quad \\
\quad \quad \text{for each } t_k \in v_{ij}, t_k \in \text{in}_\text{trans}(p) \text{ and } \text{token}_\phi(p) > 0, p \in P_i \{ \text{(3)} \\
\quad \\
\quad \quad \quad \tau = (\Sigma_{i=0}^k \alpha_i, \Sigma_{i=0}^k \beta_i); \quad \langle t_0, t_1, \ldots, t_k \rangle \text{ is a prefix of } v_{ij} \text{(4)} \\
\quad \ises_i = \text{IBS_Synthesize}(v_{ij}, t_k, \tau, \phi_i); \text{(5)} \\
\quad \text{if } M_i = \exists \square \land \text{ and } \ises_i > \text{Min}_\text{IBS}_i \text{(6)} \\
\quad \quad \text{Min}_\text{IBS}_i = \ises_i \text{; (7)} \\
\quad \text{if } M_i = \forall \square \land \text{ Old}_\text{IBS}_i = \text{Old}_\text{IBS}_i \cap \ises_i \text{; } \} \text{(8)} \\
\quad \text{if } M_i = \exists \square \land \text{ and } \text{Min}_\text{IBS}_i \neq N\_ULL \text{ \(\text{IBS}\_\text{assign}\text{(Min}_\text{IBS}_i)\); } \text{(9)} \\
\quad \text{else if } M_i = \forall \square \land \text{ Old}_\text{IBS}_i \neq N\_ULL \text{ \(\text{IBS}\_\text{assign}\text{(Old}_\text{IBS}_i)\); } \text{(10)} \\
\quad \text{return NULL; } \} \text{(11)} \\
\text{return } \tau; \text{(12)} \}
\]

consisting of a set of \emph{minimally restricted} intervals is sought because such a solution contains the \emph{maximal behavior} of the original system \(S\) that satisfies specification \(\phi\).

2 \(\phi = \forall \square \land \phi_i^*\):

A variable \(\text{Old}_\text{IBS}_i\) records the intersection of all sets of restricted transition firing intervals of \(A_i\) for satisfying \(\phi\) (Step (8)). A set intersection is performed by individual intersections of each pair of intervals \(\tau_1 = (\alpha_1, \beta_1)\) and \(\tau_2 = (\alpha_2, \beta_2)\), that is, \(\tau_1 \cap \tau_2 = (\alpha', \beta')\), where \(\alpha' = \max\{\alpha_1, \alpha_2\}\) and \(\beta' = \min\{\beta_1, \beta_2\}\).

\(\text{IBS}\_\text{assign}\) in Steps (9) and (10) assigns the final set of interval bounds to the system \(S\).

\(\text{IBS}\_\text{Synthesize}\) in Table 3 synthesizes (modifies) the firing interval bounds for a sequence of transition firings, which is a prefix of a schedule \(v_{ij} = \langle t_0, t_1, \ldots, t_k, \ldots \rangle\), such that the modified system satisfies both \(\phi\) and the aggregate delay interval \(\tau\). The switch-case statement in Step (1) to Step (7) first decides what is the least restriction on \(\tau = (\alpha, \beta)\) by calculating \(\tau' = (\alpha', \beta')\) such that \(\phi\) is satisfied. Then, depending on whether the calculated restriction
Table 3. Synthesis of Interval Bounds Set

<table>
<thead>
<tr>
<th>IBS_Synthesize ((v_{ij}, t, \tau, \phi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedule (v_{ij} = (t_{i0}, \ldots, t_{ik}, \ldots));</td>
</tr>
<tr>
<td>transition (t = t_k \in v_{ij});</td>
</tr>
<tr>
<td>FIB (\tau = (\alpha, \beta));</td>
</tr>
<tr>
<td>TRS (\phi = M\tilde{\phi}, M \in {\exists \phi_{~c}, \forall \phi_{~c}}, \tilde{\phi} = (x_1, \ldots, x_{</td>
</tr>
</tbody>
</table>

\[
\text{switch } \sim \{ \\
\text{case } <: \text{ if } (c \leq \beta) \quad \tau' = (\alpha, c - 1); \text{ break;} \\
\text{case } \leq: \text{ if } (c \leq \beta) \quad \tau' = (\alpha, c); \text{ break;} \\
\text{case } =: \quad \tau' = (c, c); \text{ break;} \\
\text{case } \geq: \text{ if } (c \geq \alpha) \quad \tau' = (c, \beta); \text{ break;} \\
\text{case } >: \text{ if } (c \geq \alpha) \quad \tau' = (c + 1, \beta); \text{ break;}
\} \quad \text{let } \tau' = (\alpha', \beta') \\
\text{if } \alpha' > \alpha \quad \text{ do} \\
\text{for } j = k, \ldots, 0 \text{ do} \{ \\
\alpha_{ij} += \text{min}\{\alpha' - \alpha, \beta_{ij} - \alpha_{ij}\}; \quad \text{(10)} \\
\text{if } (\alpha' - \alpha \leq \beta_{ij} - \alpha_{ij}) \text{ break;} \quad \text{(11)} \\
\text{else } \alpha' -= \beta_{ij} - \alpha_{ij}; \quad \text{(12)} \\
\text{if } j = 0 \text{ return Unsynthesizable; } \} \quad \text{(13)} \\
\text{if } \beta' < \beta \text{ do} \{ \quad \text{(14)} \\
\text{for } j = k, \ldots, 0 \text{ do} \{ \\
\beta_{ij} -= \text{min}\{\beta - \beta', \beta_{ij} - \alpha_{ij}\}; \quad \text{(16)} \\
\text{if } (\beta - \beta' \leq \beta_{ij} - \alpha_{ij}) \text{ break;} \quad \text{(17)} \\
\text{else } \beta' += \beta_{ij} - \alpha_{ij}; \quad \text{(18)} \\
\text{if } j = 0 \text{ return Unsynthesizable; } \} \quad \text{(19)} \\
\text{return } \{(\alpha_{ij}, \beta_{ij}) : j = 0, \ldots, k\}; \quad \text{(20)} \}
\]

is on EFT (Step (8)) or LFT (Step (14)), there is a loop for modifying the firing interval bounds \((\alpha_{ij}, \beta_{ij})\) of transitions starting from the \(k\)th one. If even after all transitions have firing intervals modified and \(\phi\) is still not satisfied then an error is returned (Steps (13) and (19)). Otherwise, the set of modified firing intervals is returned (Step (20)).

After applying the controller synthesis algorithm (Table 2) to a system \(S\), some transition firing intervals of the TFCPNs in \(S\) are restricted into smaller intervals such that the restricted (controlled) system \(S'\) satisfies a given specification TRS \(\phi\) and there is no other lesser restricted system \(S''\) that can satisfy \(\phi\). An example will be given in Section 4.
4. APPLICATION EXAMPLE

A 2-process system example is given in this section to illustrate the proposed SERTS synthesis algorithm. It consists of two TFCPN \((F_1 \text{ and } F_2)\) as shown in Fig. 3 and a Timed Reachability Specification (TRS) formula as below:

\[
\phi : \exists \bigcirc \leq 7 \langle 002 \rangle \land \exists \bigcirc \geq 30 \langle 0000001 \rangle
\]

According to our proposed algorithm (Table 1), we apply quasi-static data scheduling and controller synthesis to the given system.

**QSDS for \(F_1\):** Since \(t_{12}\) and \(t_{13}\) are conflicting transitions, two CF components \((R_{11} \text{ and } R_{12} \text{ in Fig. 4})\) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time intervals.

\[
v_{11} = (t_{11}t_{12}t_{11}t_{12}t_{14}), \quad 11 \leq \tau(v_{11}) \leq 22
\]

\[
v_{12} = (t_{11}t_{13}t_{15}t_{15}), \quad 13 \leq \tau(v_{12}) \leq 26
\]

There can be two sets of valid schedules for this TFCPN as given below.

\[
\Sigma_1 = \{(t_{11}t_{12}t_{11}t_{12}t_{14}), (t_{11}t_{13}t_{15}t_{15})\}
\]

\[
\Sigma_2 = \{(t_{11}t_{13}t_{15}t_{15}), (t_{11}t_{12}(t_{11}t_{13}t_{15}t_{15})^k t_{11}t_{12}t_{14}, \quad k \in \mathcal{N})\}
\]

**QSDS for \(F_2\):** Since \(t_{22}\) and \(t_{23}\) are conflicting transitions, two CF components \((R_{21} \text{ and } R_{22} \text{ in Fig. 5})\) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time
The set of valid schedules for this TFCPN is as given below.

\[ \Sigma_3 = \{(t_{21}t_{22}t_{24}t_{26}t_{26}t_{26}t_{26}t_{29}t_{29}t_{26}), (t_{21}t_{23}t_{25}t_{27}t_{29}t_{29}t_{26})\} \]  

**Controller Synthesis:** In Equation (2), the first conjunct in \( \phi \) corresponds to \( F_1 \) and specifies that the TFCPN \( F_1 \) reaches a marking within less than or equal to 7 time units such that there are no tokens in places \( p_1 \) and \( p_2 \) and there are two tokens in \( p_3 \). The second conjunct corresponds to \( F_2 \) and specifies that the TFCPN \( F_2 \) reaches a marking after 30 time units, inclusive, such that there are no tokens in any of the first six places \( (p_1 \ldots , p_6) \) and there is one token in place \( p_7 \). Applying the controller synthesis algorithm from Table 2, we have the following results.

**FIBS for \( F_1 \):** First, consider the conjunct in \( \phi \) that corresponds to \( F_1 \), that is, \( \exists \Theta \leq 7(002) \). Since there is only one schedule \( (v_{12} = (t_{11}t_{13}t_{15}t_{15})) \) in \( \Sigma_1 \) (Equation (4)) that results in \( p_3 \) having tokens. We calculate the time required by the prefix of the schedule that leads to 2 tokens in \( p_3 \) as follows:

\[
\begin{align*}
2 + 3 & \leq \tau(t_{11}) + \tau(t_{13}) \leq 3 + 5 \\
5 & \leq \tau(t_{11}) + \tau(t_{13}) \leq 8
\end{align*}
\]
Thus applying the IBS synthesis algorithm from Table 3, for the time spent on the schedule prefix \((t_{11}, t_{13})\) to satisfy \((\leq 7)\) constraint, the firing interval of \(t_{13}\) is modified as follows.

\[
\tau(t_{13}) = (3, 4)
\]  

\textbf{FIBS for } F_2: \text{ For TFCPN } F_2, \text{ we must consider both the schedules } v_21 \text{ and } v_22 \text{ from Equations (7) and (8) because both the schedules have prefixes that lead to a token in place } p_7. \text{ First, the aggregate delay interval is calculated for a prefix of } v_21 \text{ as follows.}

\[
25 \leq \tau(t_{21}t_{22}t_{24}t_{24}t_{26}t_{26}t_{26}t_{28}) \leq 56
\]

Thus, to satisfy the constraint of \(\geq 30\), the firing interval of \(t_8\) is modified as follows.

\[
\tau(t_{28}) = (5, 5)
\]  

When we consider a prefix of schedule \(v_{22}\), as shown below it is impossible to modify any firing interval of transitions in \(T_2\) to satisfy the \(\geq 30\) constraint because the maximum firing delay is only 28.

\[
11 \leq \tau(t_{21}t_{23}t_{25}t_{27}t_{27}t_{28}) \leq 28
\]

After modifying the firing intervals of transitions \(t_3\) and \(t_8\), we get the two controlled TFCPN as illustrated in Figs. 6 and 7.

As a last note on controller synthesis for this example, let us suppose the TRS specification is changed to the following.

\[
\phi' : \exists \Diamond_{\leq 7}(002) \land \exists \Diamond_{\geq 60}(0000001)
\]
Then, there is no modification of any firing interval of any transition that can make the system $S$ to satisfy $\phi'$. In this case, the controller is unsynthesizable.

5. CONCLUSION

Instead of ad-hoc, trial-and-error methods that engineers use in developing Soft Embedded Real-Time Systems (SERTS), it has been proposed in this work how SERTS can be developed by a formal automatic synthesis method. To satisfy the limited memory space and processor power requirements of a soft real-time embedded system, two phases, namely Quasi-Static Data Scheduling (QSDS) and Firing Interval Bound Synthesis (FIBS) are performed before code generation. Engineers will benefit from our work when he/she applies the proposed method to automatically and formally synthesize a system specification modeled as a set of TFCPNs. Future research directions include the extension of system models (TFCPN) to more general ones such that a larger domain of system can be synthesized.

REFERENCES


This page intentionally left blank.
Part Two

Distributed Systems Testing
TOWARDS A FORMAL FRAMEWORK FOR INTEROPERABILITY TESTING*

César Viho, Sébastien Barbin and Lénaïck Tanguy
IRISA/IFSIC - Université de Rennes I, Campus de Beaulieu, 35042 Rennes, France
{ viho,ltanguy,sbarbin }@irisa.fr

Abstract This paper shows how the existing concepts of conformance testing can be used to define a formal framework for interoperability testing. First, the different possible interoperability testing architectures are discussed. Then, we define several interoperability relations based on the existing relations defined for conformance testing. A comparison of these relations is given, in terms of their power to detect non-interoperability. Some guidelines are given to help in generating interoperability tests.

Keywords: interoperability, test, architecture, relation, protocol, conformance

1. INTRODUCTION

In the context of distributed systems, there are basically two approaches to testing implementations to ensure that they will work effectively together. Conformance testing determines whether a single implementation under test (IUT) conforms to its specification (generally a standard) or not. In contrast, interoperability testing determines the ability of two or more implementations to work together in a real operational environment. As we can see, the purposes of these two kinds of tests are not really the same. While conformance testing evaluates an implementation in terms of its correspondence to a specific standard, interoperability testing compares an implementation with other products.

The words "interoperability", "interwork", "interoperate" are often used in the descriptions of computer systems. Different needs of interoperability testing can be identified. The most basic is to put together two implementations built by different vendors and verify that they "interwork" correctly. Another common situation is the so called “one against N” interoperability testing in which there already exist N (N≥1)

*This work has been supported by the CELAR/TCOM Contract 9842.561 MTI (Méthodologie de test d’interopéabilité), the European ITEA Project 99011 RTIPA, and the CNRS/Programme Télécommunications 99N36/0029
systems working together. One may wonder if a new system S will inter-operate with the N existing systems. In other words, the question here is “will the N+1 systems still work together?”. There exist many other situations which require interoperability. Thus, it is difficult to give a unique definition of this notion. Indeed, several definitions of interoperability exist [1, 2, 3, 4]. However, the functional meaning is clear: the components of the system under test (SUT) have to communicate with each other correctly and provide the expected services.

A lot of work have been done in the area of conformance testing [5] leading to a precise definition of conformance [6]. The so called conformance/implementation relations in [7, 8] give formal characterizations of conditions under which an IUT can be considered as conformant to its specification. This allows the automatic generation of conformance tests [9]. Unfortunately, such work is lacking for interoperability testing [3]. There are several reasons for this situation. The first reason is because conformance of implementations has been considered a prerequisite for achieving interoperability. Thus, a very important effort has been concentrated on conformance testing. Another reason is that interoperability testing is considered a pragmatic and practical requirement, as it relates to implementations; contrary to conformance testing where the specification serves as reference.

The fact that both conformance and interoperability testing concerns the same objects (specification, IUT, etc) is evident. Thus, most of the concepts, methodologies and theory developed for conformance testing can be adjusted for interoperability testing. This simple observation suggests to adapt the existing formal conformance testing framework for interoperability testing. Following this idea, the work presented in this paper proposes a formal framework (architectures, interoperability relations, tests generation) for interoperability testing.

This paper is structured as follows. Section 2 presents the different classes of interoperability testing architectures. For each class, contexts in which an architecture can be used are explained. We give an indication of how the existing architectures can be positioned in regard to these classes. The model and notations used are described in Section 3. Section 4 gives formal definitions of interoperability relations, that are based on the existing relations defined for conformance testing. Each interoperability relation specifies formally the conditions to be satisfied by two implementations in order to be considered interoperable. These relations are compared in terms of their power to detect non interoperability. In Section 5, we give guidelines to generate interoperability tests for the proposed architectures and relations. Conclusion and future work are in Section 6.
2. INTEROPERABILITY ARCHITECTURES

Depending on how the implementations to be tested are interconnected, and depending on the degree to which we can observe their interaction, different interoperability testing architectures can be used. Indeed, a lot of architectures have been proposed [1, 2, 4, 10].

2.1. Definition of testing architectures

Let us consider the general architecture of figure 1 where the SUT is composed of two IUTs ($IUT_1$ and $IUT_2$). Each IUT is a black-box. The Upper Interface $UI_i$ (resp. Lower Interface $LI_i$) is the interface of the $IUT_i$, through which it communicates with its upper (resp. lower) layer. The expected services are furnished through $UI_i$ while $LI_i$ is used by $IUT_i$ to interact with the peer entities in the same layer.

Depending on the environment in which the interoperability testing will be done, different levels of control and observation of the SUT are possible. Thus, the Test System (TS) in charge of testing, via its PCOs (points of control and observation), the interoperability of this SUT can consist of some or all of these following elements. The Upper Tester $UT_i$ (resp. Lower Tester $LT_i$) is part of TS in charge of the control and/or the observation of $UI_i$ (resp. $LI_i$), via the Upper PCO $UP_i$ (resp. the Lower PCO $LP_i$). The Tester $T_i$ (composed by $UT_i$ and/or $LT_i$) is the part of TS in charge of the control and/or observation of $IUT_i$.

The different possible compositions of both TS and SUT induce different kinds of interoperability testing architectures as described below. The reader can see that they are easily extendible to the general case where TS and/or SUT are composed by more than two components.

2.1.1 Unilateral interoperability testing architectures.
These architectures correspond to the situation in which the control and/or observation of only one component ($IUT_1$ or $IUT_2$) of the SUT
is possible. Thus, the decision of interoperability is based on these partial observations done by only one tester $T_1$ or $T_2$. The tester $T_i$ may contain only $LT_i$ (resp. $UT_i$). This can happen if the upper (resp. lower) interface is embedded under an upper (resp. lower) layer protocol. In this case, the architecture is called Unilateral Lower (resp. Upper) Interoperability Testing Architecture. The Unilateral Total Interoperability Testing Architecture corresponds to the case where the tester $T_i$ contains both $UT_i$ and $LT_i$. In this case, a test coordination procedure (TCP) may exist between $UT_i$ and $LT_i$.

These architectures are used when the other IUTs which compose the SUT are embedded (i.e., all or part of their interfaces are not accessible) or in a “one against N” interoperability testing context (see section 1).

2.1.2 Bilateral interoperability testing architectures.
In these architectures, each tester $T_i$ realizes separately the control and observation of the corresponding $IUT_i$, using a Unilateral Interoperability Testing Architecture. Depending on the composition of the testers, the terms Upper, Lower and Total apply also here.

This architecture is interesting in the unusual situations where the designers of the IUTs require their own control and opinion of how their respective IUT interact with the others. This happens for example when a certain level of confidentiality is still required by each of the vendors [4].

2.1.3 Global interoperability testing architectures.
In these architectures, the control and observation are done globally by the two testers $T_1$ and $T_2$ on both sides ($IUT_1$ and $IUT_2$) of the SUT. They are called global because the decision of interoperability is achieved according to a global view (simultaneously on both sides) of the SUT. This is the main difference with the previous architectures, as a test coordination procedure exists between the testers. In this sense, it is the more complex architecture. Global upper, lower, and total architectures are defined as for bilateral architectures.

2.1.4 Hybrid interoperability testing architectures.
Some situations of interoperability combine previously defined architectures. All the architectures which are not strictly unilateral, bilateral or global architectures are called hybrid architectures. As an example, the architecture in which we have only the upper tester in tester $T_1$ and upper and lower testers of the tester $T_2$, is typically a hybrid architecture. This kind of architecture can be encountered in the “testing against reference” context where one of the implementations serves as the reference, the upper tester over the tested IUT is used as a responder.
2.2 Positioning the existing architectures

Many interoperability architectures and associated terminologies can be found in the literature [1, 2, 4, 10]. In the following, we give elements which help in understanding why there is so much confusion in the state of the art of interoperability testing architectures. We show that each of the existing architectures can be associated with one of the canonical architectures described above.

Black-box versus grey-box testing
It is well-known that conformance testing is done in a black-box context. But there is still a debate about considering interoperability testing as black-box or grey-box testing [2]. The contested point comes from the possibility for the interoperability testing architecture to observe or not (and/or control) the interactions between the implementations which compose the SUT. Our purpose is not to close the debate, but one can see that “grey-box” architectures correspond to architectures which include lower testers. In contrast, the so called “black-box” architectures are those which do not contain any lower tester.

Protocol versus service interoperability testing
The definition of interoperability can be “layer-oriented” [4, 11]. For example in [4], they consider four levels of interoperability called respectively protocol, service, application and user-perceived interoperability architectures. It is easy to see that protocol interoperability architectures correspond to a lower interoperability testing architecture, while service (application or user perceived) interoperability architectures can be definitely classified as upper interoperability testing architectures.

Active versus passive testing architectures
If the test system has the possibility to give stimuli (control) and/or to corrupt events in the SUT, the corresponding testing architectures are called active [1]. If not, they are called passive architectures [10, 11]. Thus, any possible architecture (including those proposed in this paper) can be considered either as active or passive.

Monitor versus arbiter
In these architectures [10], a test component is placed between the implementations to control (monitor) or observe (arbiter) their communication. To be monitor or arbiter depends on the active/passive behavior of the tester. Thus, any architecture proposed in this paper that contains at least one lower tester can be used as monitor or arbiter. Arbiters are used to identify faulty implementations [12].
3. MODELS AND NOTATIONS

In order to give formal definitions of the notion of interoperability, we need a model which allows the formal description of all the elements involved in the interoperability testing activity. As said before, most of the terms and components in interoperability testing are similar to those used in conformance testing. We choose the commonly used model of the IOLTS (Input-Output Labeled Transition System) [13] to model specifications as well as implementations.

3.1. IOLTS, definitions and notations

Definition 3.1 An IOLTS is a tuple \( M = (Q^M, \Sigma^M, \Delta^M, q_0^M) \) where

- \( Q^M \) is the set of states of the system and \( q_0^M \) is the initial state.
- \( \Sigma^M \) denotes the set of observable (input and/or output) events on the interaction points (with the environment) of the system. Thus, we have: \( \Sigma^M \subseteq P^M \times \{?,!\} \times A^M \) where \( P^M \) is the finite set of interaction points (ports) through which the system communicates with its environment (lower or upper layer, or other systems), ""?"" and ""!"" respectively denote a input and an output of message, \( A^M \) is the alphabet of input-output messages exchanged by the system through its ports.
- \( \Delta^M \subseteq Q^M \times (\Sigma^M \cup \tau) \times Q^M \) is the transition relation, where \( \tau \in A^M \) denotes an internal event. We note \( q \xrightarrow{\alpha}_{M} q' \) for \( (q, \alpha, q') \in \Delta^M \).

\[\begin{array}{c}
\text{Figure 2. A specification } S \text{ and possible implementations } I_1, I_1', I_2, I_2', I_2''
\end{array}\]

\( \Sigma^M \) can be decomposed as follow: \( \Sigma^M = \Sigma^M_U \cup \Sigma^M_L \), where \( \Sigma^M_U \) (resp. \( \Sigma^M_L \)) is the set of messages exchanged on the upper (resp. lower) interface. \( \Sigma^M \) can be also decomposed in order to distinguish input messages from output messages. \( \Sigma^M = \Sigma^M_I \cup \Sigma^M_O \) where \( \Sigma^M_I \) (resp. \( \Sigma^M_O \)) is the finite set of input (resp. output) messages.

In the following we denote the set of all input/output labeled transition systems by \( IOLTS \). Let us consider an IOLTS \( M \in IOLTS \), and let \( \alpha \in \Sigma^M \) with \( \alpha = p.\{?!,\}.m, \mu_i \in \Sigma^M \cup \tau, \sigma \in (\Sigma^M)^* \), \( q, q', q_i \in Q^M \):
\[ \begin{align*}
\bullet & \text{ out}(q) = \Delta \{ \alpha \in \Sigma^M_\sigma | \exists \ q' \text{ and } q \overset{\alpha}{\Rightarrow}_M q' \} \text{ is the set of outputs from } q. \\
\bullet & \text{ q after } \sigma = \Delta \{ q' \in \Theta^M | q \overset{\sigma}{\Rightarrow}_M q' \} \text{ is the set of states which can be reached from } q \text{ by the sequence of actions } \sigma. \text{ By extension, all the states reached from the initial state of the IOLTS } M \text{ is (q}_0^M \text{ after } \sigma) \text{ and will be noted by } (M \text{ after } \sigma). \text{ In the same way, Out}(M, \sigma) = \Delta \text{ out}(M \text{ after } \sigma).
\end{align*} \]

In interoperability testing, we usually need to observe some specific events among all the possible traces of a SUT. These traces, reduced to the expected messages, can be obtained by a projection of those traces on a set representing criteria used to select the expected events.

**Definition 3.2** Let us consider an IOLTS M, a trace \( \sigma \in (\Sigma^M)^* \), \( \alpha \in \Sigma^M_\sigma \), and a set X. The projection of \( \sigma \) on X is noted by \( \sigma/X \) and is defined by: \( \epsilon/X = \epsilon, (\alpha.\sigma)/X = \sigma/X \) if \( \alpha \notin X \), and \( (\alpha.\sigma)/X = \alpha.(\sigma/X) \) if \( \alpha \in X \).

For example, consider a trace \( \sigma = U?A.l!a.l?c.U!B \in (\Sigma^M)^* \), then \( \sigma/\{l!a, U?D\} = l!a, \sigma/\Sigma^M_\sigma = U?A.U!B, \sigma/\Sigma^M_\sigma = U?A.l?c \). Notice that \( \sigma/X \in (\Sigma^M)^* \) but we do not have necessarily \( \sigma/X \in \text{Traces}(M) \).

**Definition 3.3** Let us consider an IOLTS M, a trace \( \sigma \in (\Sigma^M)^* \) and a set X, then \( \text{Out}_X(M, \sigma) = \Delta \text{ Out}(M, \sigma) \cap X \) is the set of outputs from \( M \text{ after } \sigma \), reduced to the outputs that belong to X.

### 3.2. Synchronous and asynchronous interaction

**Definition 3.4** (Synchronous composition | |) The synchronous composition of two IOLTS \( M_1 \) and \( M_2 \) is noted \( M_1 | | M_2 = (Q^{M_1} \times Q^{M_2}, \Sigma^{M_1} | | \Sigma^{M_2}, \Delta^{M_1} | | \Sigma^{M_2}, \Delta^{M_1} | | \Sigma^{M_2}) \) where \( \Sigma^{M_1} | | \Sigma^{M_2} \subseteq \Sigma^{M_1} \cup \Sigma^{M_2} \), and the transition relation \( \Delta^{M_1} | | \Sigma^{M_2} \) is obtained as follows, \( \forall (q_1, q_2) \in Q^{M_1} \times Q^{M_2}, \)

\[
\begin{align*}
(q_1, a, q_1') & \in \Delta^{M_1} a \in \Sigma^{M_1} \cup \{ \tau \}, (q_2, a, q_2') \in \Delta^{M_2} a \in \Sigma^{M_2} \cup \{ \tau \}, \\
((q_1, q_2), a, (q_1', q_2')) & \in \Delta^{M_1} | | \Delta^{M_2} (q_1, q_2) \in \Delta^{M_1} | | \Sigma^{M_2}, ((q_1, q_2), a, (q_1', q_2')) \in \Delta^{M_1} | | \Delta^{M_2} \quad (1)
\end{align*}
\]

\[
\begin{align*}
(q_1, a, q_1') & \in \Delta^{M_1} a \in \Sigma^{M_1} \cup \{ \tau \}, (q_2, a, q_2') \in \Delta^{M_2} a \in \Sigma^{M_2} \cup \{ \tau \}, \\
(q_1, q_2) & \in \Delta^{M_1} | | \Delta^{M_2} (q_1, q_2) \in \Delta^{M_1} | | \Delta^{M_2}, (q_1, q_2) \in \Delta^{M_1} | | \Sigma^{M_2} \quad (2)
\end{align*}
\]
The interaction between components of a distributed system is done through an environment. This interaction can be either synchronous, or asynchronous. As in [13], we will suppose that the (synchronous or asynchronous) environment can be modeled by an IOLTS $\mathcal{E}$. Thus, the interaction of an IOLTS $S_i$ with another IOLTS $S_j$ through an environment $\mathcal{E}$ is obtained by the synchronous composition $S_i \parallel \mathcal{E} \parallel S_j$. This will be noted by $S_i \parallel S_j$ and corresponds to the reachability graph in [1, 2].

4. INTEROPERABILITY RELATIONS

In this section, we give a formal definition of the notion of interoperability based on the model of IOLTS introduced in section 3.1. To a certain extent, testing the interoperability between two implementations consists of a kind of conformance testing of their interaction with the expected behavior or service. Thus, it becomes natural to study how these conformance relations [7, 8] can be adapted to obtain interoperability relations. We will consider the most commonly used $\text{ioconf}$ conformance relation [9, 13]. It states that an implementation $I$ is conformant to its specification $S$ if after a trace of $S$, outputs of $I$ are foreseen in $S$.

**Definition 4.1** $I \text{ ioconf } S = \Delta \forall \sigma \in \text{Traces}(S), \text{Out}(I, \sigma) \subseteq \text{Out}(S, \sigma)$

4.1. The notion of interoperability relation

An interoperability relation formally specifies conditions to be satisfied by two implementations in order to be considered interoperable.

**Definition 4.2** An interoperability relation is a relation $R$ between two implementations $I_1$ and $I_2$: $R(I_1, I_2)$ means that $I_1$ interoperates with $I_2$.

Interoperability relations are not transitive: the fact that an implementation $I_1$ interoperates with $I_2$ and that $I_2$ interoperates with $I_3$, does not allow one to draw a conclusion regarding interoperability between $I_1$ and $I_3$. The main difference between an interoperability relation and a conformance relation is that: a conformance relation connects an implementation with its specification while an interoperability relation concerns implementations. More precisely, two implementations can interoperate even though their interaction does not correspond to anything allowed in their respective specification. Thus, an interoperability relation can take into account or not the specifications on which the implementations are based. This leads to two main classes of interoperability relations called in the following "Specification-less" interoperability relations and "Specification-based" interoperability relations.

As the "Specification-less" interoperability relations do not use any specification, it is impossible to generate $a$ priori tests based on these
relations. Verifying these relations can be done preliminary to other tests using specification-based relations. This can explain why it is called interconnectivity testing in [4]. As a consequence, we will no longer discuss these relations in the sequel.

4.2. Specification-based relations

Specification-based interoperability relations are relations which refers to the specification of at least one component of the SUT.

4.2.1 Specification-based lower interoperability relations.

In this section, we focus on the lower layer interfaces of the implementations. We consider the different possible situations and we propose the associated interoperability relations which can be used.

The relation $R_1$ considers the situation where we have the specification of only one of the two implementations. Let us consider that $S_1$ is the specification used to develop the implementation $I_1$. Now, let us consider that only the lower interface of $I_1$ is accessible. During the interaction between $I_1$ and $I_2$, the least we can expect from the implementation $I_1$ is to behave as foreseen in its specification $S_1$. This is described in the relation $R_1$. It considers the traces observed $\left(\text{Traces}(I_1 \| I_2)\right)$ during the interaction between $I_1$ and $I_2$. It states that, after any corresponding trace in the specification $S_1$ of the unilaterally considered implementation (here $I_1$), all the outputs (sent to the other IUT on the lower layer) are allowed in $S_1$.

**Definition 4.3 (Unilateral Lower Interoperability Relation $R_1$)**

\[
\mathcal{R}_1(I_1, I_2) =_{\Delta} \forall \sigma_1 \in \text{Traces}(S_1), \forall \sigma_2 \in \text{Traces}(I_1 \| I_2), \sigma/\Sigma^{I_1} = \sigma_1 \Rightarrow \text{Out}_{\Sigma^{I_1}}(I_1 \| I_2, \sigma) \subseteq \text{Out}_{\Sigma^{S_1}}(S_1, \sigma_1)
\]

**Remarks:**
- In figure 2, $R_1(I_1, I_2)$ but $\neg R_1(I_2, I_1)$.
- Notice also that there is a subtle difference with the $\text{ioconf}$ relation: $R_1(I_x, I_y)$ does not mean necessarily that $I_x \text{ ioconf } S_x$.

The relation $R_1$ can be applied independently for $I_2$ (based on the specification $S_2$). When we have both $R_1(I_1, I_2)$ and $R_1(I_2, I_1)$, this corresponds to the interoperability relation $R_2$ defined below.

**Definition 4.4 (Bilateral Lower Interoperability Relation $R_2$)**

\[
\mathcal{R}_2(I_1, I_2) =_{\Delta} \mathcal{R}_1(I_1, I_2) \land \mathcal{R}_1(I_2, I_1)
\]

In figure 2, $R_2(I_1, I_2)$ but $\neg R_2(I_1, I_2')$. In the two relations $R_1$ and $R_2$, only lower layer outputs of each IUT are considered. Typically, we can have $R_2(I_1, I_2)$, whatever happens in their interaction with upper layers.
The next relation $R_3$ is called **global** because it is based on the global behavior of the interactions between respectively the two specification-s $(S_1 \parallel S_2)$ and the two implementations $(I_1 \parallel I_2)$. The relation $R_3$ states that: after a trace included in the composition of the specifications $S_1 \parallel S_2$ and observed in the interaction between the two implementations, all outputs observed on the lower layer are allowed in $S_1 \parallel S_2$.

**Definition 4.5 (Global Lower Interoperability Relation $R_3$)**

$$R_3 (I_1, I_2) = \Delta \forall \sigma \in \text{Traces}(S_1 \parallel S_2) \Rightarrow \text{Out}_{\Sigma_L^{I_1 \parallel I_2}} (I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma_L^{S_1 \parallel S_2}} (S_1 \parallel S_2, \sigma).$$

In figure 2, $R_3 (I_1, I_2)$ but $\neg R_3 (I_1, I_2)$. $R_3$ is a kind of reduced conformance relation in the sense that only a subset of outputs is considered.

### 4.2.2 Specification-based upper interoperability relations.

The following three relations are similar to the previous ones. The difference is that only observations on the upper (instead of lower) layer are used. Consequently, similar explanations and remarks apply for the three relations which are enumerated below.

- The **Unilateral Upper Interoperability relation** $R_4$ can be defined as follows: $\mathcal{R}_4(I_1, I_2) = \Delta \forall \sigma_1 \in \text{Traces}(S_1), \forall \sigma \in \text{Traces}(I_1 \parallel I_2), \sigma / \Sigma_{I_1} = \sigma_1 \Rightarrow \text{Out}_{\Sigma_U^{I_1 \parallel I_2}} (I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma_U^{S_1 \parallel S_2}} (S_1 \parallel S_2, \sigma_1)$.

- For the **Bilateral Upper Interoperability relation** $R_5$, we have: $\mathcal{R}_5(I_1, I_2) = \Delta \mathcal{R}_4(I_1, I_2) \land \mathcal{R}_4(I_2, I_1)$.

- Finally, the **Global Upper Interoperability relation** $R_6$ is defined by: $\mathcal{R}_6(I_1, I_2) = \Delta \forall \sigma \in \text{Traces}(S_1 \parallel S_2), \text{Out}_{\Sigma_U^{I_1 \parallel I_2}} (I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma_U^{S_1 \parallel S_2}} (S_1 \parallel S_2, \sigma)$.

On the example of figure 2, we have:

- $\mathcal{R}_4(I_1, I_2)''$, $\mathcal{R}_4(I_2, I_1)$, $\mathcal{R}_4(I_2'' , I_1)$ but $\neg \mathcal{R}_4(I_1, I_2)$.

- $\mathcal{R}_5(I_1, I_2)$, $\mathcal{R}_5(I_2, I_1)$ but $\neg \mathcal{R}_5(I_1, I_2)$.

- $\mathcal{R}_6(I_1, I_2)''$, $\mathcal{R}_6(I_1, I_2)$, $\mathcal{R}_6(I_1, I_2)''$ $\mathcal{R}_6(I_1' , I_2)$, but $\neg \mathcal{R}_6(I_1, I_2)$.

### 4.2.3 Specification-based total interoperability relations.

The following relations are qualified **total** because the decision of interoperability uses observations on both upper and lower layers. These observations can be done locally/unilaterally, bilaterally or globally. As in the two sections above, we obtain the three following relations.

- The **Unilateral Total Interoperability relation** $R_7$ is defined as follows: $\mathcal{R}_7(I_1, I_2) = \Delta \forall \sigma_1 \in \text{Traces}(S_1), \forall \sigma \in \text{Traces}(I_1 \parallel I_2), \sigma / \Sigma_{S_1} = \sigma_1 \Rightarrow \text{Out}_{\Sigma_{I_1 \parallel I_2}} (I_1 \parallel I_2, \sigma) \subseteq \text{Out}(S_1, \sigma_1)$.

- It is easy to obtain the **Bilateral Total Interoperability relation** $R_8$. It is defined by: $R_8(I_1, I_2) = \Delta R_7(I_1, I_2) \land R_7(I_2, I_1)$.

- The definition of the **Global Total Interoperability relation** $R_9$ is
also straightforward. \( R_9(I_1, I_2) = \Delta \forall \sigma \in \text{Traces}(S_1 \parallel S_2), \ Out(I_1 \parallel I_2, \sigma) \subseteq \text{Out}(S_1 \parallel S_2, \sigma) \).

Remarks:

- In figure 2, we have \( R_7(I_1, I_2') \) but \( \neg R_7(I_1, I_2) \), \( R_8(I_1, I_2') \) but \( \neg R_8(I_1, I_2) \) and \( \neg R_8(I_1, I_2') \).
- One can notice that \( R_7(I_1, I_2) = R_1(I_1, I_2) \land R_4(I_1, I_2), \ R_8(I_1, I_2) = R_2(I_1, I_2) \land R_5(I_1, I_2), \) and \( R_9(I_1, I_2) = R_3(I_1, I_2) \land R_6(I_1, I_2) \).
- \( R_9(I_x, I_y) \) corresponds to \( I_x \parallel I_y \text{ ioconf } S_x \parallel S_y \). We will discuss this point later in Section 5.

4.3. Comparison of interoperability relations

According to the rigor required for interoperability testing, we need to know which interoperability relation to use. In this section, we give a comparison between the specification-based interoperability relations defined in Sections 4.2.1, 4.2.2 and 4.2.3, in terms of their power of non-interoperability detection. Let us call the set of these relations \( \text{SBIR} \), then the formalization of the comparison of the relations leads to the well-known theory of testing equivalence [6, 13] and the related preorder expressed here by a relation \( \leq_R \subseteq \text{SBIR} \times \text{SBIR} \).

**Definition 4.6** \( \forall R_x, R_y \in \text{SBIR}, \ R_x \equiv_R R_y \Rightarrow \forall I_1, I_2 \in \text{IO LTS}, \ R_x(I_1, I_2) \Rightarrow R_y(I_1, I_2) \)

So, \( R_x \leq_R R_y \) means that interoperability tests based on \( R_x \) detect more non-interoperable implementations than \( R_y \). By extension, the testing equivalence between two relations will be denoted by \( \equiv_R \). We note \( R_x \not\leq_R R_y \) to say that \( R_x \) and \( R_y \) are not comparable. It is represented by “-” in Figure 3 which synthesizes all the comparison of the specification-based interoperability relations. Proofs are given below.

![Figure 3. Comparison of specification-based interoperability relations](image)

4.3.1 Some proofs. The decision of interoperability relies on observations done on the available interfaces. So the set of interfaces can be used to compare interoperability relations. “Lower-oriented” and
“upper-oriented” relations are not comparable because their sets of interfaces are not comparable. Total interoperability relations are stronger than the corresponding upper or lower interoperability relations because their set of observations is greater. Bilateral and Global relations are stronger than their corresponding unilateral relations. Most of the proofs are based on these remarks. Nevertheless, we have observed that complete formal proofs of $R_2 \cong R_3$, $R_5 \cong R_6$, and $R_8 \cong R_9$ need some intermediate lemmas and propositions. We give them in the following.

**Lemma 4.1** Let $M_1, M_2 \in \mathcal{IOLTS}$, and $\sigma \in \text{Traces}(M_1 \parallel M_2)$,

$\text{Out}(M_1 \parallel M_2, \sigma) = \text{Out}(M_1, \sigma/\Sigma^{M_1}) \cup \text{Out}(M_2, \sigma/\Sigma^{M_2})$.

**Proof:** Let $(q_1, q_2) \in ((M_1 \parallel M_2)$ after $\sigma)$ and $a \in \text{Out}(M_1 \parallel M_2, \sigma)$. Rules of definition 3.4 apply for $(q_1, q_2)$. Thus, we have either $a \in \Sigma^{M_1}$ or $a \in \Sigma^{M_2}$ which means that $a \in \text{Out}(M_1, \sigma/\Sigma^{M_1}) \cup \text{Out}(M_2, \sigma/\Sigma^{M_2})$. In the other sense, it is easy to see that $\text{Out}(M_1, \sigma/\Sigma^{M_1}) \cup \text{Out}(M_2, \sigma/\Sigma^{M_2}) \subseteq \text{Out}(M_1 \parallel M_2, \sigma)$. Thus, we have $\text{Out}(M_1 \parallel M_2, \sigma) = \text{Out}(M_1, \sigma/\Sigma^{M_1}) \cup \text{Out}(M_2, \sigma/\Sigma^{M_2})$. ◊

**Lemma 4.2** Let $M_1, M_2 \in \mathcal{IOLTS}$, and $\sigma \in \text{Traces}(M_1 \parallel M_2)$,

$\text{Out}_{\Sigma^{M_1} \parallel M_2}(M_1 \parallel M_2, \sigma) = \text{Out}_{\Sigma^{M_1} \parallel M_2}(M_1 \parallel M_2, \sigma)$

**Proof:** Using the definition 3.4 of the composition $\parallel$, we have $\Sigma^{M_1} \parallel M_2 \subseteq \Sigma^{M_1} \cup \Sigma^{M_2} \Rightarrow \text{Out}_{\Sigma^{M_1} \parallel M_2}(M_1 \parallel M_2, \sigma) \subseteq \text{Out}_{\Sigma^{M_1} \parallel \Sigma^{M_2}}(M_1 \parallel M_2, \sigma)$. Now, let $a \in \text{Out}_{\Sigma^{M_1} \parallel \Sigma^{M_2}}(M_1 \parallel M_2, \sigma)$, then $a \in \text{Traces}(M_1 \parallel M_2)$. Thus, $a \in \Sigma^{M_1} \parallel M_2$ and $\text{Out}_{\Sigma^{M_1} \parallel \Sigma^{M_2}}(M_1 \parallel M_2, \sigma) \subseteq \text{Out}_{\Sigma^{M_1} \parallel \Sigma^{M_2}}(M_1 \parallel M_2, \sigma)$. ◊

**Proposition 4.1** $R_2 \subseteq R_3$.

**Proof:** Let $I_1, I_2, S_1, S_2 \in \mathcal{IOLTS}$ such that $R_2(I_1, I_2)$. Let $\sigma \in \text{Traces}(S_1 \parallel S_2)$ such that $\sigma \in \text{Traces}(I_1 \parallel I_2)$. Let us consider $\sigma_1 = \sigma/\Sigma^{I_1}$ and $\sigma_2 = \sigma/\Sigma^{I_2}$, we can notice that $\sigma_1 \in \text{Traces}(S_1)$ and $\sigma_2 \in \text{Traces}(S_2)$. Using the definition of $R_2(I_1, I_2)$, we have $\text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_1, \sigma_1) \cup \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_2, \sigma_2)$. Thus, with the definition 3.3, and the classical properties of $\cup$ and $\cap$, we have :

$\text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_1, \sigma_1) \cup \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_2, \sigma_2)$.

Using the definition of $\sigma_1$ and $\sigma_2$, we have :

$\text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_1, \sigma_1/\Sigma^{I_1}) \cup \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_2, \sigma_2/\Sigma^{I_2})$

$\Leftrightarrow \text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{I_1} \parallel I_2}(S_1 \parallel S_2, \sigma)$ (lemma 4.1)

$\Leftrightarrow \text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{S_1} \parallel S_2}(S_1 \parallel S_2, \sigma)$ (lemma 4.2)

Thus, $\forall \sigma \in \text{Traces}(S_1 \parallel S_2)$, $\text{Out}_{\Sigma^{I_1} \parallel I_2}(I_1 \parallel I_2, \sigma) \subseteq \text{Out}_{\Sigma^{S_1} \parallel S_2}(S_1 \parallel S_2, \sigma)$

which proves: $R_2 \subseteq R_3$. ◊
Proposition 4.2 $R_3 \subseteq R_2$.

Proof: Let $I_1, I_2, S_1, S_2 \in IOLTS$ such that $R_3(I_1, I_2)$ and let us prove that it implies $R_1(I_1, I_2)$. As $R_1(I_1, I_2)$ does not refer to the specification of $I_2$, we can consider that $S_2 = I_2$. Let $\sigma_1 \in Traces(S_1)$, and $\sigma \in Traces(I_1 \parallel I_2), \sigma/\Sigma_1 = \sigma_1$. As $\sigma_1 \in Traces(S_1)$ and $S_2 = I_2$, we have $\sigma \in Traces(S_1 || S_2)$. Using the fact that $R_3(I_1, I_2)$ gives $Out_{S_1 || S_2}(I_1 || I_2, \sigma) \subseteq Out_{S_1}(S_1 || S_2, \sigma)$. We want to prove that $Out_{S_1}(I_1 || I_2, \sigma) \subseteq Out_{S_2}(S_1 || S_2, \sigma)$.

As $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$ and considering only the outputs of $I_1$, we have: $Out_{S_1}(I_1 || I_2, \sigma) \subseteq Out_{S_1}(S_1 || S_2, \sigma)$. Using the definition 3.3, lemmas 4.1 and 4.2 gives the following equations $Out_{S_1}(S_1 || S_2, \sigma) = Out_{S_1}(I_1 || I_2, \sigma) \subseteq Out_{S_1}(S_1 || S_2, \sigma)$. Thus, $Out_{S_1}(I_1 || I_2, \sigma) \subseteq Out_{S_1}(S_1 || S_2, \sigma)$. So, $\forall \sigma_1 \in Traces(S_1), \forall \sigma \in Traces(I_1 || I_2), \sigma/\Sigma_1 = \sigma_1$, $Out_{S_1}(I_1 || I_2, \sigma) \subseteq Out_{S_1}(S_1 || S_2, \sigma)$ and $R_3(I_1, I_2) \Rightarrow R_1(I_1, I_2)$.

Using the fact that $R_3$ is symmetrical, we have $R_3(I_1, I_2) \Rightarrow R_1(I_2, I_1)$. Thus, $R_3(I_1, I_2) \Rightarrow (R_1(I_1, I_2) \wedge R_1(I_2, I_1)) = R_2(I_1, I_2)$. 

Theorem 4.1 $R_3 \equiv_R R_2$, $R_6 \equiv_R R_5$, and $R_9 \equiv_R R_8$

Proof: The first proof is achieved with the proposition 4.1 and the proposition 4.2. The two other proofs are achieved with similar lemmas adapted to upper and total contexts. This means that the global and bilateral interoperability relations are equivalent.

5. INTEROPERABILITY TESTS GENERATION

Several methods have been proposed to generate interoperability tests [2, 14, 15, 16]. Now that we have identified the interoperability architectures and we have defined possible interoperability relations, we give in this section some guidelines to derive interoperability tests.

How to choose interoperability relations?

Depending on several parameters (accessibility of interfaces of the implementations, problems of confidentiality, etc), different architectures can be used for interoperability testing. In Section 2.1, we have identified the four possible classes of interoperability testing architectures. An architecture determines the possible interoperability relations which can be used. As proved in Section 4.3, the power of the generated tests depends on the chosen relation. In the following, we give elements which help in choosing appropriate interoperability relations.

For unilateral testing architectures, the choice of interoperability relations depends on available interfaces (upper, lower or both).
As bilateral and global interoperability relations are equivalent (see theorem 4.1), it can be more interesting to use a bilateral interoperability relation in a global testing architecture. Indeed, this choice avoids the often difficult and error-prone task of designing test coordination procedures which are required when using global interoperability relations.

Hybrid architectures can be tested by generalizing interoperability relations to hybrid cases, or by using a combination of existing relations. For example, architectures where one lower interface is unavailable can be tested using either a combination of $R_4$ (unilateral upper) relation and $R_7$ (unilateral total) relation, or a combination of $R_6$ and $R_1$ (global upper and unilateral lower) relations.

Can conformance tests be used for interoperability?

There is still considerable debate about the value of conformance testing as a means of achieving interoperability [1, 16]. Let us consider the $\text{ioconf}$ relation. Proposition 5.1 suggests that an implementation tested with conformance relation $\text{ioconf}$ does not need to be tested again when the $R_7$ interoperability relation is used.

**Proposition 5.1** Let $I_1, I_2 \in IOLTS$, $I_1 \text{ioconf} S_1 \Rightarrow R_7(I_1, I_2)$

*Proof:* Let us consider $I_1, I_2 \in IOLTS$ and $S_1 \in IOLTS$ the specification on which $I_1$ is based. $I_1 \text{ioconf} S_1$ implies $\forall \sigma_1 \in \text{Traces}(S_1), \text{Out}(I_1, \sigma_1) \subseteq \text{Out}(S_1, \sigma_1)$. Let us consider a trace $\sigma \in \text{Traces}(I_1 \parallel I_2)$ such that $\sigma/I_1 = \sigma_1$. Using lemma 4.1 and properties of the projection, we have $\text{Out}_{I_2}(I_1 \parallel I_2, \sigma) = \text{Out}(I_1, \sigma_1)$. Thus, as $\text{Out}(I_1, \sigma_1) \subseteq \text{Out}(S_1, \sigma_1)$, we have $I_1 \text{ioconf} S_1 \Rightarrow R_7(I_1, I_2)$. ◊

The problem is that conformance testing is not exhaustive. Thus, the result of Proposition 5.1 is “theoretical”. On the other hand, proposition 5.2 says that an implementation can be considered $R_7$-interoperable with another one without being conformant to its specification.

**Proposition 5.2** Let $I_1, I_2 \in IOLTS$, $\mathcal{R}_7(I_1, I_2) \not\Rightarrow I_1 \text{ioconf} S_1$

*Proof:* It is based on the fact that $\text{Traces}(I_1 \parallel I_2)/\Sigma^{I_1} \subset \text{Traces}(I_1)$. Thus, it can exist a trace $\sigma'_1 \in \text{Traces}(I_1 \parallel I_2)/\Sigma^{I_1}$, such that $R_7(I_1, I_2)$ but not $I_1 \text{ioconf} S_1$. ◊

We have similar results for other interoperability relations. Given the complexity of standards and the limits on exhaustive conformance testing, interoperability testing is seen to be a practical requirement. In particular, it is used to uncover incompatibilities (such as incompatible options, coding, etc) even when both implementations have successfully undergone (whatever rigorous) conformance tests. It is a matter of increasing the confidence in the real interoperability between implementations.
How to generate interoperability tests

Interoperability relations defined in Section 4.2 can be written in the shape of “parameterized” conformance relations. Indeed, interoperability relations consider the interaction between implementations, rather than a separately considered implementation in conformance relations. Thus, parameters to be introduced in conformance relations (like `io-conf`) in order to obtain interoperability relations are implementations (resp. specifications) with which the considered implementation (resp. specification) has to interact and the available interfaces. Thus, one may wonder if existing automatic test generation tools like TGV [9] can be used and/or adapted for automatically generating interoperability tests. The general answer to this question needs some further

Theorems 4.1 which prove that bilateral and global interoperability relations are equivalent hints that we may avoid the construction of the composition of the specifications. It is well-known that this construction (even if it is done on-the-fly like in TGV) is a bottleneck for most of the existing tools. This is our current work and first results are promising.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a formal framework (architectures, interoperability relations, test generation techniques) for interoperability testing. We show that the existing concepts of conformance testing can be used to give a formal definition of the notion of interoperability. Different interoperability testing architectures are proposed and discussed. We have defined several interoperability relations based on the existing implementation relations defined for conformance testing. A comparison of these relations is given, in terms of their power to detect non-interoperability. This comparison suggests that the generation of interoperability tests do not necessarily need to build the whole interaction between the specifications of the implementations. This paper ends with some guidelines to help in generating interoperability tests.

As future work, we will focus on quiescence management in the interoperability relations. Based on the defined interoperability relations and the results of their comparison presented in this paper, we will investigate more deeply the different methods and associated algorithms in order to automatically and efficiently generate interoperability tests.

REFERENCES


Acknowledgments
Authors wish to thank their colleagues D. Clarke for carefully reading this paper, and C. Jard & T. Jéron for their constructive remarks.
DISTRIBUTED TEST USING LOGICAL CLOCK

Young Joon Choi¹, Hee Yong Youn², Soonuk Seol¹, and Sang Jo Yoo³
¹School of Engineering, Information and Communications University
{cyjnice, suseol}@icu.ac.kr
²School of ECE, Sungkyunkwan University, Suwon, Korea
youn@ece.skku.ac.kr
³The Graduate School of Information Technology & Telecommunications, Inha University, Inchon, Korea
sjyoo@inha.ac.kr

Abstract

It is difficult to test a distributed system because of the task of controlling concurrent events. Existing works do not propose the test sequence generation algorithm in a formal way and the amount of message is large due to synchronization. In this paper, we propose a formal test sequence generation algorithm using logical clock to control concurrent events. It can solve the control-observation problem and makes the test results reproducible. It also provides a generic solution such that the algorithm can be used for any possible communication paradigm. In distributed test, the number of channels among the testers increases non-linearly with the number of distributed objects. We propose a new remote test architecture for solving this problem. SDL Tool is used to verify the correctness of the proposed algorithm, and it is applied to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection as a case study.

Keywords: concurrent events, control-observation problem distributed testing, logical clock, output-shifting faults, test sequence

1. INTRODUCTION

Distributed objects have been implemented by Open Distributed Processing (ODP) [11] and others. ODP is an ISO/ITU standard, offering a generic framework for the development of distributed system. In this paper, distributed system developed based on ODP is taken as the target [1] to test.
Benattou et al. [1] proposed a method to test distributed system as a black box using a distributed test method. Here extra signals were proposed to be used among the testers through coordination channels, called multicast channel, for synchronizing them. The method has a merit of making the events occurred at the distributed objects be totally ordered with special signals, but has a drawback of possible output-shifting faults [8]. A method in [2] is to use synchronization messages for controlling concurrent events as a grey box. However, it has a difficulty in choosing a way to observe the internal status of implementation under test (IUT) and needs other messages in addition to TS. Luo et al. [3] analyzed existing models based on synchronized TS and surveyed fault coverage in terms of output, transfer, and hybrid. However, it did not propose any Test Sequence (TS) generation algorithm in a formal way.

In overall, existing works for controlling concurrent events in the test of distributed system do not propose the algorithm in a formal way and increase the amount of message within IUT due to synchronization. To overcome the drawbacks, we propose a TS generation algorithm using logical clock that formally generates signals to control concurrent events by numerically labeling the events of TS with logical clock values and comparing them. It can also solve the control-observation problem [1,3]. The proposed TS generation algorithm based on logical clock allow the test results to be reproduced since a totally ordered TS can be achieved with the controlling of concurrent events.

In distributed test, several testers are used and synchronized by exchanging coordination messages. While traditional distributed test method is effective in testing IUT of multiple ports, it has a shortcoming that the number of channels among the testers increases non-linearly with the number of distributed objects. To overcome this drawback, we also propose an approach of remote test architecture. Specification and Description Language (SDL) Tool [14] is used to verify the correctness of the proposed algorithm, and it is applied to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection as a case study.

The rest of the paper is structured as follows. Section 2 explains the background on testing distributed system. The proposed TS generation algorithm based on logical clock is proposed in Section 3. Section 4 applies the algorithm to the remote test architecture, and finally Section 5 concludes the paper with some future research work.

2. BACKGROUND

When a distributed system is tested, some of the events executed by the distributed objects do not always have casual relationship but possibly
concurrent relationship with each other. If the events are concurrent, the following **control-observation problem** may take place [1, 3].

- When concurrent inputs are applied to a black box of a form of IUT, the test results depend on the order of input process in the IUT.
- In testing a black box IUT, even though the mapping of an input and its output is incorrect, test results may be correct.

The events occurred at each distributed object, which is an element of a distributed system, may have concurrent relationship with other events. In this case, the test results can be different from each other when an IUT is tested with the same TS repeatedly since all the events are not totally ordered. Therefore we need a mechanism to make concurrent events have causal relation in order to reproduce the same test results. Causal relation is also called “happened before” relation, represented by ‘→’. Relation ‘a’ → ‘b’ should satisfy one of the following rules to be ‘happened before’ events [5, 6].

**Rule 1**

(1) ‘a’ and ‘b’ are in the same distributed object and ‘a’ → ‘b’.
(2) ‘a’ is sender and ‘b’ is receiver. However, in case of synchronous communication, both send and receive events occur at the same time.
(3) ‘a’ → ‘c’ and ‘c’ → ‘b’.

Logical clocking is a time-stamping mechanism, which can be represented with linear time, vector time, and matrix time [5]. In this paper, we adopt vector time describing logical clock in one-dimensional matrix whose field is assigned to each object [5]. The value of a field of logical clock denotes the time the corresponding event occurs at a distributed object. With the vector time, the following rule should be satisfied so that two events can be causally related.

**Rule 2**

If the following relation is satisfied between event ‘X’ and ‘Y’ in an IUT consisting of distributed objects a, b and c, the two events are said to have a causal relation “a → b” except a case as \((X_a = Y_a, X_b = Y_b, X_c = Y_c)\). The logical clock values of events ‘X’ and ‘Y’ are \((X_a, X_b, X_c)\) and \((Y_a, Y_b, Y_c)\), respectively.

\[
(X_a \leq Y_a, X_b \leq Y_b, X_c \leq Y_c)
\]

A stable state means a global state, which is not changed into another state without any input [7]. In testing a black box IUT, it is difficult to observe the transient states and only the stable states are visible.
Figure 1 represents a finite state machine (FSM), which reaches stable state-3 via transient state-1 and 2 after input ‘a’ is applied at stable state-0. A tester can infer the test result by observing state-0 before the input ‘a’ was applied and state-3 after the output ‘x’ and ‘z’ are obtained. This allow to reduce the number of states which should be managed in the test and also avoid the difficulty of observing transient states.

3. THE PROPOSED ALGORITHM

In this section we first analyze the existing method proposed for controlling concurrent events in a black box IUT. We then propose a TS generation algorithm based on logical clock.

3.1 The Problems in Existing Test Method

As a case study, we consider a typical test method reported in [1]. Figure 2(a) represents a FSM M, where each state is stable. The FSM M is a 3-port FSM whose inputs and outputs at each port are shown in Table 1. Figure 2(b) is faulty FSM M derived from Figure 2(a). Here each port is connected to a tester whose identifier is the same as the port.

Table 1. Inputs and outputs at each port of Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>port 1</th>
<th>port 2</th>
<th>port 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>a</td>
<td>b</td>
<td>c, d</td>
</tr>
<tr>
<td>Output</td>
<td>x, w</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

Definition 1: Transition T is represented by \((s_i, s_j, x/y)\), where \(x \in X, y \in Y, \{s_i, s_j\} \subseteq S, s_j = \delta(s_i, x)\), and \(y = \lambda(s_i, x)\). X is a finite set of input symbols, Y is a finite set of output symbols, S is a finite set of states, \(\delta\) is a state transfer function, and \(\lambda\) is an output function.
If an IUT implemented with a faulty FSM M is tested by a TS (T1 → T3 → T2 → T5 → T6), the control-observation problem may occur in the subsequence T3 → T2 and T5 → T6. None of the testers can detect the faults if ‘b’ is read first by the IUT when the input ‘c’ of transition T3 and the input ‘b’ of transition T2 are applied to the IUT concurrently in state-1. It is because the order of the outputs observed by the testers is correct even though the mapping of the input and its output is incorrect in the subsequence T5 → T6.

The TS of a distributed object usually consists of inputs and outputs, and signal ‘C’ and ‘O’ controlling concurrent events. The signal ‘C’ gives casual relation to concurrent inputs applied to IUT, while signal ‘O’ gives causal relation between concurrent input and output. The following steps produce signal ‘C’ and ‘O’. A new TS for each distributed object including signal ‘C’ and ‘O’ is called Local Test Sequence (LTS).

1. Label logical clock values for the events of TS.
2. Classify the events into either causal or concurrent by comparing the logical clock values of the events.
3. If there exist concurrent events, generate signal ‘C’ and ‘O’ which make the concurrent events be causally related and insert them to the TS of each distributed object.

To solve the control-observation problem, [1] proposed coordination channels whose function is to send or receive signal ‘C’ and ‘O’ among the testers. However, output-shifting fault which is a consequence of the control-observation problem can occur.

Refer to Figure 3(a) where an IUT is tested by TS (a/xyz → b/xz). The signal ‘C’ and ‘O’ are not generated by the method proposed in [1]. As a result, none of the testers can detect the faults even if the output ‘z’ of “a/xyz” is shifted forward and the output ‘x’ of “b/xz” is shifted backward as shown in Figure 3(b). This is because the order of the input and output observed by each tester is the same as Figure 3(a). This is called output-shifting faults [8]. Output-shifting faults are the case that the test results are correct even though the mappings of input and output are incorrect. This occurs since it cannot properly control the testers receiving the outputs of the

![Figure 3. An example of output-shifting faults.](image)
previous transition which has concurrency relation with the new input. A new formal way for signal generation is thus needed to be developed.

3.2 The Proposed Local Test Sequence Generation Algorithm

Here we propose an LTS generation algorithm which can solve the control-observation problem including output-shifting faults in a formal way using logical clock. We make assumptions as follows:

- IUT is a black box.
- The logical clock vector is represented as a format (tester 1, IUT, tester 2, tester 3). When an event occurs at an element, the value of the corresponding field is increased.
- The output receive delay for a transmission is zero.
- Logical clock is increased by the unit time.

Figure 4 shows TS1 (T1 → T3 → T2 → T4 → T7) which tests the IUT implemented with FSM M. Observe that each event is labeled with a logical clock value. To control concurrent events classified by Rule 2, extra signals are needed. Table 2 lists the comparisons of logical clock values labeled with the events of subsequence T1 → T3 and T3 → T2. Here Compare_LC(x, y) is a function that returns 'true' if event 'x' and 'y' satisfies Rule 2. Otherwise, it returns 'false'.

Figure 4. Logical clock labeling at the events of Test Sequence 1 (TS1).

Table 2. Comparison of logical clock values.

<table>
<thead>
<tr>
<th>Event Sequence</th>
<th>Logical Clock Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (0, 1, a/xz) → T3 (1, 1, c/xz)</td>
<td>Compare_LC(a, c) = true, Compare_LC(x, c) = false, Compare_LC(z, c) = true</td>
</tr>
<tr>
<td>T3 (1, 1, c/xz) → T2 (1, 1, b/wy)</td>
<td>Compare_LC(c, b) = false, Compare_LC(x, b) = false, Compare_LC(z, b) = false</td>
</tr>
</tbody>
</table>
According to Table 2, only output ‘x’ of transition T1 and input ‘c’ of transition T3 are concurrent in subsequence T1 → T3. In this case, a signal making input ‘c’ and output ‘x’ be casually related is needed. This signal is called signal ‘O’. Accordingly, Tester 3 that provides input ‘c’ of transition T3 to IUT sends signal ‘O_{3,1}’ to Tester 1, which receives output ‘x’ of transition T1. Similarly, input ‘b’ of transition T2 is concurrent with all the inputs and outputs of transition T3 in subsequence T3 → T2. Therefore a signal is needed to let input ‘b’ be applied to the IUT after transition T3 occurs. This signal is called signal ‘C’. In this case, Tester 3 that is one of the testers receiving the outputs of transition T3 sends signal ‘C_{3,2}’ to Tester 2, which applies input ‘b’ of transition T2 to the IUT. The signal ‘C_{3,2}’ suggests that Tester 3 had already received output ‘z’. However, Tester 2 sends additional signal ‘O_{2,1}’ to Tester 1 since input ‘b’ is still concurrent with output ‘x’ of transition T3.

None of the testers can detect faults even though output ‘y’ is backward-shifted since the test scenario of Tester 2 is the same as that for correctly implemented FSM being tested by subsequence T4 → T7. To solve this problem, Tester 3 sends signal ‘O_{3,2}’ to Tester 2 before Tester 3 applies input ‘d’ to the IUT.

As aforementioned, the output receive delay is assumed to be zero, and thus each tester receives the output of a transition instantly. Therefore, signal ‘O’ controlling concurrent outputs is generated by the tester which sends input to the IUT. This assumption was also adopted in [1]. With this assumption, though, the TS generation algorithm has some limitations in testing real distributed system.

For example, refer to Figure 5. Tester 1 decides that the IUT is faulty when it receives output ‘x’ in “a/xz” after Tester 1 receives signal ‘O_{3,1}’ even if the IUT is correctly implemented. Also Tester 2 cannot detect faults in case that output ‘y’ in “c/xy” is backward-shifted. This is because synchronization among the testers is done through synchronization signal, but IUT cannot be directly synchronized with the testers. Accordingly, the testers should have sufficient waiting time to synchronize with the IUT.

![Figure 5. Waiting time and over-waiting time problem.](image-url)
However, if Tester 1 receives backward-shifted output ‘x’ in “c/xy” after
some waiting time, it is impossible to detect the fault. Let us call this over-
waiting problem. In order to prevent the problem, the waiting time and
receive delay time of outputs for a transmission are assumed to be zero. The
related definitions and the proposed algorithm are presented as follows.

**Definition 2**

- In transition $T = (s_i, s_j, x/y)$, “x/y” stands for “!x?y”.
- ‘-’ means that signal is sent to the tester and ‘+’ means that signal is
  received from other testers.
- Port(x) is a function that returns ‘port id’ generating input or output ‘x’.
- PortS(X) = {Port(x1), Port(x2), …, Port(xn)}, where X = {x1, x2, …, xn} and xi is an event.
- Compare_LCS(x, Y) is a function that returns false if Compare_LC(x, a)
  = false for all a ∈ Y, and return true otherwise where Y is a set of input or
  output.
- Label_LC(x, k) is a function that updates logical clock value of port k
  and records logical clock value of x where x is an input or output.
- Label_LCS(X, PortS(X)) = {Label_LC(x1, Port(x1)), Label_LC(x2,
  Port(x2)), …, Label_LC(xn, Port(xn))}. Append(X, Y) is a function
  concatenating string X and Y.
- Append_In (Port(x), x) = Append(LTSPort(x), Append(“!”, x)) where
  LTSPort(x) is an LTS of the tester connected to Port(x).
- Append_Out(PortS(X), X) = {Append(LTSPort(X1), Append(“?”, “x1”)),
  Append(LTSPort(X2), Append(“?”, “x2”)), …, Append(LTSPort(Xn),
  Append(“?” , “xn”))}.

**Algorithm LTS** /* LTS GENERATING ALGORITHM */

1: INPUT: TS = $l_0; Y_0 \ldots, l_{n-1}; Y_n$ /*The elements of $Y_i$ can be more than zero and is represented with an array */
2: OUTPUT: local test sequences = (LTS$_0$, …, LTS$_n$) /* n is the number of distributed objects/
3: Send_Set = null /* Send_Set is a set of testers receiving signal ‘O’ */
4: for(i = 1; i ≤ m; i++) /* m is the number of transitions */
5: Label_LC(s_i, Port(a)); /* Logical clock values labeling at input s_i and outputs Y */
6: Label_LCS(Y, PortS(Y));
7: end for
8: for(i = 1; i ≤ m; i++)
9: k = Port(s_i); h = Port(s_{i+1});
10: Append_In(Port(s_i), s_i);
11: if i < m
12: case |Y_i| ≠ 0:
13: Append_Out(PortS(Y_i), Y_i); /* Comparison of logical clock values for generating signal ‘C’ */
14: if (Compare_LCS(s_i+1, Y_i) = false) and (Compare_LC(s_{i+2}, s_i) = false)
15: select a sender (tester) which generates signal ‘C’ where sender ∈ PortS(Y);
16: Append(LTSPort, Append(“!”, “C_sended”)); Append(LTSPort, Append(“+”, “C_sended”));
17: end if
18: if (Compare_LCS(Y_i, Y_{i+1}) = false) and (Port(Y_{i+1}) ≠ sender)
19: Send_Set ← Send_Set ∪ Port(Y_{i+1}); /* output-shifting faults */
20: end if
21: if i < m
22: end for
23: for(i = 0; i < |Y_i|; i++) /* Comparison of logical clock values for generating
24: if (Compare_LCS(Y_i[i], Y) = true) and (Port(Y_i[i]) ∈ PortS(Y)) and
   (Port(Y_{i+1}[i] ≠ h)) or (Compare_LCS(Y_{i+1}[i], Y) = false) and (Port(Y_{i+1}[i] ≠ h))
25: Send_Set ← Send_Set ∪ Port(Y_{i+1}[i]);
26: end if
In the algorithm above, Lines from 4 to 7 are to label logical clock values for the events of TS. Whenever input or output is generated from the IUT, Label_LC (or Label_LCS) function updates the logical clock value of the distributed object where the input or output is generated. Lines from 14 to 40 are to generate signal ‘C’ and ‘O’ in the transitions except the final transition. When the output of a transition is not null, the logical clock values of the events between the current transition and the input events of the next transition are compared using Line 19. If they are concurrent, signal ‘C’ is generated. Lines 19 and 24 are the statements comparing logical clock values for controlling forward output-shifting faults and backward output-shifting faults, respectively. When the output of a transition is null, the logical clock values between the current and next input are compared using Line 29. If they are concurrent, signal ‘C’ is generated. Line 33 is to solve backward output-shifting faults in case of null output by comparing the clock values. Figure 6 is a test scenario obtained as a result of applying Algorithm_LTS to TS1.

![Figure 6. Applying Algorithm_LTS to TS1.](image)
Note that state 0, 1 and 2 in Figure 6 are stable states since they are not changed to other states before a new input is applied to the IUT. If the area between the stable states is represented as a block, happened-before relationships also exist between the blocks since the events in each block are totally ordered with the events in other blocks by signal ‘C’ and ‘O’.

3.3 Extension of the Proposed Local Test Sequence Generation Algorithm

The LTS generation algorithm [1] and the proposed Algorithm_LTS assume that the output receive delay is zero. If the assumption is not satisfied, test results may not be able to be reproduced. For example, in Figure 6, even though IUT is correct, the test results can be faulty when Tester 3 receives output ‘z’ and then sends signal ‘O_{3,1}’ to Tester 1 before output ‘x’ is received by Tester 1 in the first block. To solve the problem, LTS for Tester 3 should be updated so that input ‘c’ can be sent to the IUT after Tester 1 sends signal ‘O_{3,1}’ to Tester 3 and Tester 3 receives both signal ‘O_{3,1}’ and output ‘z’.

Let us call the LTS generation algorithm for which output receive delay is not zero Algorithm_LTS_DELAY. Append_RD(X, Y) is a function attaching Y to X while keeping the order of the elements of them. For example, Append_RD(“xy”, “z”) means that ‘xyz’, ‘xzy’, or ‘zxy’.

```
FUNCTION: Algorithm_LTS_DELAY
1: INPUT: TS = s_0; Y_1, ..., s_n; Y_m/* The elements for Y_i can be more than zero and is represented by an array */
2: OUTPUT: TS = s_0; Y_1, ..., s_n; Y_m/* n is the number of distributed objects */
3: Send_Set = null /* Send_Set is a set of testers which should receive signal ‘O’ */
4: for (i = 1; i <= n; i++) /* m is the number of transitions */
5: label LCS(i, Ports(Y_i));
6: /* Logical clock values labeling at input s_i and outputs Y_i */
7: end for
8: for (i = 1; i <= m; i++)
9: k = Ports(s_i); h = Ports(s_{i+1});
10: Append_In(k, s_i);
11: if i < m
12: case Y_i != 0;
13: if (Compare_LCS(s_{i+1}, Y_i) = false) and (Compare_LCS(s_{i+1}, s_i) = false)
14: Append_Out(PortS(Y)_j, Y_i);
15: select a sender (tester) which generates signal ‘C’ where sender e PortS(Y)_j;
16: AppendLTSender, Append(“C", “sender_h”));
17: AppendRD(temp, Append("C", “sender_h”));
18: Temp Set = Ports(Y)_j sender; /* is a difference set */
19: else if (Compare_LCS(s_{i+1}, Y_i) = false) and (Compare_LCS(s_{i+1}, s_i) = true)
20: Append_Out(PortS(Y)_j, Y_i);
21: else if Compare_LCS(s_{i+1}, Y_i) = true
22: /* Case 2: comparison of logical clock values for generating signal ‘O’ */
23: for (j = 0; j < |Y_i|; j++)
24: if Compare_LC(s_{i+1}, Y_{j[j]} = false)
25: Temp Set = Temp Set ∪ Port(Y_{j[j]});
26: AppendLTSender, Append("O", Y_{j[j]}));
27: else Append(temp, Append("O", Y_{j[j]}));
28: end if
29: end for
```
The lines from 13 to 34 are modified from Algorithm_LTS. Line 13 (Case 1), Line 19 (Case 2), and Line 22 (Case 3) compare logical clock values for controlling forward output-shifting faults in case that the outputs are not null. Case 1 is applied when all the events of the current transition and an input of the subsequent transition are concurrent, Case 2 is applied when outputs of the current transition and an input of the subsequent transition are concurrent, and Case 3 is applied when some outputs of the current transition and an input of subsequent transition are causally related, respectively. The signal ‘O’ generated by each case is sent from the tester (Temp_Set), which receives the outputs of the current transition, to a tester which sends an input of the subsequent transition to the IUT. Therefore, the LTS for input-side testers should manage all combinations of the receive order of the input events. Generating signal ‘O’ that controls backward output-shifting faults is the same as the one in Algorithm_LTS.

Figure 7 is a sample test scenario. Here Specification and Description Language (SDL) Tool [14] is used to verify the correctness of Algorithm_LTS_DELAY. An FSM representing the IUT and the LTS obtained by Algorithm_LTS_DELAY for TS1 are described in SDL processes, respectively. Note that the output receive delay is not zero.
We analyze the results of Message Sequence Chart (MSC) Trace which is the test scenario for Figure 7 using reachability tree. Figure 8 shows the reachability tree obtained by the SDL Tool. The states represented at each point are global ones showing the states of Tester 1, IUT, Tester 2 and Tester 3, respectively.

The box shown in Figure 8 represents the reachability tree for transition T1. To send input ‘c’ of transition T3 to the IUT, the logical clock value should be (3, 4, 0, 3). In other words, input ‘c’ is applied to the IUT only at stable state (3, 4, 0, 3). This demonstrates that Algorithm_LTS_DELAY can reproduce the test results by controlling concurrent events.
We apply Algorithm_LTS and Algorithm_LTS_DELAY to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection. Figure 9 illustrates a message exchange where call initiator is S_A and call receivers are R_A and R_B [12, 13]. In here, the IUT has 3 ports connected to S_A, R_A and R_B.

![Figure 9. Message exchange for the establishment of Q.2971 network-side npoint-to-multipoint call/connection.](image)

In Figure 10(a), the testers must receive the outputs without delay. However, message exchange in communication networks such as Q.2971 cannot satisfy such condition. The test scenario of Figure 10(b) can be said to be much more general. Table 3 compares the LTS generation algorithms: the algorithm in [1], Algorithm_LTS, and Algorithm_LTS_DELAY.

![Figure 10. The LTS obtained by the two algorithms.](image)

**Table 3. Comparison of Local Test Sequence generation algorithms.**

<table>
<thead>
<tr>
<th></th>
<th>[1]</th>
<th>Algorithm_LTS</th>
<th>LTS_DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output receive delay</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Backward output-shifting faults</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Forward output-shifting faults</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Over-waiting time problem exists</td>
<td>-</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Channel complexity</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
</tbody>
</table>
In Table 3, Algorithm_LTS prevents output-shifting faults except over-waiting time problem. With Algorithm_LTS_DELAY, appropriate waiting time is allowed for preventing over-waiting time problem. The channel complexity of the algorithms is $O(n^2)$ since they employ a test method of mesh-structure. The numbers of signal ‘C’ and ‘O’ generated by the proposed algorithms are same, while it is slightly larger than that of [1].

4. TEST ARCHITECTURE

In this section, we propose a method for testing distributed system using Algorithm_LTS_DELAY with no coordination channels.

4.1 The Proposed Test Architecture

The number of coordination channels among the testers in distributed test environment exponentially increases since the channels are connected as a mesh. So, the diagnostic power of a tester diminishes due to the communication overhead. In order to solve the problem, a test architecture having $O(n)$ channel complexity is proposed as Figure 11. Here Management and Control Entity (MCE) receives signal ‘O’ from the testers and sends the signal to them.

![Proposed test architecture.](image)

Here the transitions (state a, state b, x, y) and (state b, state c, x, y), and the tester connected to Port(x) should receive synchronization signals except cases that Port(y) = null, Port(x) = Port(x) and Port(x) = Port(y). This is because the testers are synchronized with other testers via only MCE. Therefore, only signal ‘O’ is used to synchronize the testers. Figure 12 shows the result obtained by applying the proposed algorithm to the Figure 9.
To apply the proposed algorithm, extra signals are needed between the testers and an MCE in case of the transitions having concurrent events or no output for any input. To generate signal ‘O<sub>M,3</sub>’ to prevent backward output-shifting faults for output ‘SETUP’ in “ADD PARTY/SETUP”, extra signal ‘O<sub>1,M</sub>’ is generated as shown in a box of Figure 12. Note that only MCE can transfer signal ‘O’. Here, signal ‘O<sub>M,1</sub>’ is synchronization signal of Tester 1, while Tester 1 applies input ‘ADD PARTY’ to the IUT.

The test architecture suggested in Figure 11 is more effective than conventional distributed test architecture for testing Q.2971. It is because modification of test architecture is minimal and test errors caused by channel faults becomes less while the number of call receives increases. However, there may be a bottleneck in MCE due to the increase of signal ‘O’s.

5. CONCLUSION AND FUTURE WORK

In this paper, a LTS generation algorithm has been proposed to test a distributed system using logical clock. Proposed algorithm classifies the events into concurrent or causal by labeling and comparing the logical clock values of the events of TS. Based on that, it generates additional signals ‘C’ and ‘O’ to control concurrent events and inserts them to the TS for each distributed object.

We have also proposed a variation of the LTS generation algorithm for which the output receive delay can be nonzero in distributed test architecture with channels among testers. It can automatically produce LTS onto already generated TS. Analyzing test scenario given with the proposed algorithm using reachability tree demonstrates that the proposed algorithm can solve the control-observation problem including output-shifting faults in a formal way.
In conventional distributed test architecture, the number of channel increases non-linearly since channels among testers are connected as a mesh. To overcome this drawback, a test architecture having O(n) channel complexity is proposed where the testers are not directly connected with each other. With this, diagnostic power of the testers can increase owing to the simplified test architecture.

As future work, over-waiting problem and appropriate ways to test distributed system using a dynamic testing method will be studied.

REFERENCES

DIAGNOSING MULTIPLE FAULTS IN COMMUNICATING FINITE STATE MACHINES

Khaled El-Fakih+, Nina Yevtushenko++ and Gregor v. Bochmann+
+School of Information Technology and Engineering, University of Ottawa, ON, K1N 6N5, Canada. {kelfakih, bochmann} @ site.uottawa.ca
++Tomsk State University, 36 Lenin Str., Tomsk, 634050, Russia, yevtushenko.rff@elefot.tsu.ru

Abstract
In this paper, we propose a method for diagnostic test derivation when the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine can be faulty. The method enables to decide if it is possible to identify the faulty machine in the system, once faults have been detected in a system implementation. If this is possible, it also provides tests for locating the faulty component machine. Two examples are used to demonstrate the different steps of the method. The method can also be used for locating faults within a machine when the system specification and implementation are given in the form of a single FSM.

Keywords: Diagnostics, conformance testing, communicating finite state machines

1. INTRODUCTION

The purpose of conformance testing is to check whether an implementation conforms to its specification. Usually a conforming implementation is required to have the same I/O behavior. An interesting complementary yet more complex problem is to locate the differences between a specification and its implementation when the implementation is found to be nonconforming [7]. A solution to this problem has various applications. For example, it makes easy the job of correcting the implementation so that it conforms to its specification [7].

In the software domain where a system may be represented as an FSM, some work has already been done for the diagnostic and fault localization problems.
problems [2][4][7]. However, the work done for distributed systems represented as two Communicating FSMs [3] makes some important simplifying assumptions. In [2][7] and [3] the difference between the system specification and its implementation is located under the assumption of a single fault in the implementation. In [4] the differences can be located for multiple faults under the assumption that each of the faults is reachable through non-faulty transitions.

In this paper, we consider a system consisting of two communicating FSMs, called components. One component, called context machine, communicates with the environment and the other component, called embedded machine. The interaction between these two components is assumed to be hidden, that is, unobservable. We assume that multiple output or transfer faults may occur in at most one component [5]. None of these faults will increase the number of states in the implementation of the component. It is not always possible to locate the faulty component of the given system when faults have been detected in a system implementation (SUT). This happens when certain faults in an implementation of the context and other faults in the implementation of the embedded component may cause the same observable behavior of the SUT. We present a novel approach for diagnostic test derivation. The approach enables us to decide whether it is possible to identify the faulty component in the given system, and if this is possible then tests for locating the fault are derived. We use a non-deterministic FSM for the compact representation of possible faulty transitions. The same technique can also be used for locating multiple faults when the system specification and implementation are given in the form of a single FSM.

This paper is organized as follows. Section 2 comprises necessary definitions for communicating FSMs. In Section 3, the diagnostic problem is discussed, while the method to solve the problem is presented in Section 5. In Section 4, the different steps of the method are presented illustrated by a working example. Future work is described in Section 6.

2. FINITE STATE MACHINES

A non-deterministic finite state machine (FSM) is an initialized non-deterministic Mealy machine which can be formally defined as follows [9]. A non-deterministic finite state machine $\text{A}$ is a 6-tuple $\langle S, I, O, h, D_{A}, s_0 \rangle$, where $S$ is a finite nonempty set of $n$ states with $s_0$ as the initial state; $I$ and $O$ are input and output alphabets; $D_{A}$ is a specification domain which is a subset of $S \times I$; and $h: D_{A} \rightarrow 2^{S \times O} \setminus \emptyset$ is a behavior function where $2^{S \times O}$ is the set of all subsets of the set $S \times O$. The behavior function defines the possible transitions of the machine. Given present state $s_i$ and input symbol $i$, each
pair \((s_j, o)\) ∈ \(h(s_i, i)\) represents a possible transition to the next state \(s_j\) with the output \(o\). This is also written as a transition of the form \(s_i \xrightarrow{i/o} s_j\). If \(D_A = S \times I\) then \(A\) is said to be a complete FSM; otherwise, it is called a partial FSM. In the complete FSM we omit the specification domain \(D_A\), i.e. complete FSM is 5-tuple \(A = \langle S, I, O, h, s_0 \rangle\). If for each pair \(s_i \in D_A\) it holds that \(|h(s, i)| = 1\) then FSM \(A\) is said to be deterministic. In the deterministic FSM \(A\) instead of behavior function \(h\) we use two functions, transition function \(\delta: S \times I \rightarrow S\) and output function \(\lambda: S \times I \rightarrow O\). FSM \(B = (S', I, O, g, s_0)\), \(S' \subseteq S\), is a sub-machine of complete FSM \(A\) if for each pair \(s_i \in S' \times I\), it holds that \(g(s, i) \subseteq h(s, i)\). Similar to [6], we further denote \(\text{Sub}(A)\) the set of all deterministic sub-machines of FSM \(A\). We also use the definition of a deterministic path in FSM \(A\) since only such paths can occur in its deterministic sub-machines. A deterministic path \(P\) of FSM \(A\) starts at the initial state and has no transitions with different next states and/or outputs for the same state-input combination.

As usual, function \(h\) can be extended to the set \(I^*\) of finite input sequences. Given state \(s \in S\) and input sequence \(\alpha = i_1 i_2 ... i_k \in I^*\), output sequence \(o_1 o_2 ... o_k \in h(s, \alpha)\) if there exist states \(s_1 = s, s_2, \ldots, s_k, s_{k+1}\) such that \((s_{j+1}, o_j) \in h(s_j, i_j), j = 1, \ldots, k\). We let the set \(h^o(s, \alpha) = \{\gamma | \exists s' \in S [(s', \gamma) \in h(s, \alpha)]\}\) denote the output projection of \(h\), while denoting \(h^s(s, \alpha) = \{s'| \exists \gamma \in Y^* [(s', \gamma) \in h(s, \alpha)]\}\) the state projection of \(h\). Input/Output sequence \(i_1 i_2 o_2 ... i_k o_k\) is called a trace of \(A\) if \(o_1 o_2 ... o_k \in h^o(s_0 i_1 i_2 ... i_k)\). We also use the notation \(h^s \gamma(s, \alpha)\) to denote the set \(\{s'| (s', \gamma) \in h(s, \alpha)\}\) of all states where the sequence \(\alpha\) can take FSM \(A\) from the initial state with the output response \(\gamma\). For appropriate \((s, \gamma)\) the set \(h^s \gamma(s, \alpha)\) can be empty. If the set \(h^s \gamma(s, \alpha)\) has the unique state \(s\) then we say the state \(s\) is observably reachable from the initial state via the trace \(\alpha/\gamma\).

Given states \(s_1\) and \(s_2\) of complete FSM \(A\), states \(s_1\) and \(s_2\) are equivalent, written \(s_1 \equiv s_2\), if for each input sequence \(i_1 i_2 ... i_k \in I^*\), it holds that \(h(s_1, i_1 i_2 ... i_k) = h(s_2, i_1 i_2 ... i_k)\). If states \(s_1\) and \(s_2\) are not equivalent then they are distinguishable, written \(s_1 \neq s_2\). Given complete FSM \(A\), sequence \(\alpha \in I^*\) such that \(h(s_1, \alpha) \neq h(s_2, \alpha)\) is said to distinguish states \(s_1\) and \(s_2\). FSM \(A\) with pairwise distinguishable states is called a reduced FSM.

Complete FSMS \(A = (S, I, O, h, s_0)\) and \(B = (T, I, O, g, t_0)\) are equivalent, written \(A \equiv B\), if their sets of traces coincide. It is well known, given complete deterministic FSM \(A\), there always exists a reduced FSM that is equivalent to \(A\).
2.1 A System of Two Communicating FSMs

Many complex systems are typically specified as a collection of communicating components. We consider here a special case, where the system consists of two Communicating FSMs (ComFSMs), called embedded machine (or $M_2$) and context machine (or $M_1$). We let the alphabet $X$ and $Y$ represent the externally observable input/output actions of the system, while the $U$ and $Z$ alphabets represent the internal (hidden) input/output interactions between the two components. The two (deterministic) FSMs communicate asynchronously via bounded input queues where input/output messages are stored. An FSM produces an output in response to each input. We assume that the system at hand has at most one message in transit, i.e. the next external input is submitted to the system only after it produced an external output $y$ to the previous input. Under these assumptions, the collective behavior of the two communicating FSMs can be described by product machine. The product machine $M_1 \times M_2$ is a Labeled Transition System (LTS) that describes the joint behavior of the component machines in terms of all actions within the system. If the product machine has a cycle labeled only with internal actions from the alphabet $U \cup Z$ then the system falls into live-lock when an input sequence leading to this cycle is applied; i.e. the system will not produce any external output. Here we accept a catastrophic interpretation of live-locks and, similar to [9], say the composed FSM of $M_1$ and $M_2$ is not defined. Otherwise, a composed machine, denoted as Reference System (RS) = $M_1 \odot M_2$, can be obtained from the product machine by hiding all internal actions in the product machine, and pairing input with output actions [9].

As an example, we consider the two machines $M_1$ and $M_2$ shown in Fig. 1. The set of external inputs is $X= \{x_1, x_2\}$, the set of external outputs is $Y= \{y_1, y_2, y_3\}$, the set of internal inputs is $U= \{u_1, u_2\}$, and the set of internal outputs is $Z= \{z_1, z_2\}$. Their corresponding reference system $RS = M_1 \odot M_2$ is shown in Fig. 2.

A tester, implements a given test by executing external input sequences (test cases) simultaneously against both the SUT consisting of the implementation of $M_1$ and $M_2$, and the reference system in order to generate
the observed and expected outputs. If for each test case, the sequences of the observed and expected outputs coincide then the system is said to pass the test suite.

Figure 2. Reference System ($M_1 \diamond M_2$) of the $M_1$ and $M_2$ of Fig. 1

2.2 A Fault Model for the System of Communicating FSMs

We consider a fault model based on output and transfer faults of a deterministic FSM [2]. Given complete deterministic specification and implementation FSMs $〈S, I, O, δ, λ, s_0〉$ and $〈S, I, O, Δ, Λ, s_0〉$, we say a transition $(s,i)$ has an output fault if $δ(s,i) = Δ(s,i)$ while $λ(s,i) ≠ Λ(s,i)$. An implementation has a single output fault if one and only one of its transitions has an output fault. We say that a transition has a transfer fault if $δ(s,i) ≠ Δ(s,i)$, i.e. given state $s$ and input $i$, the implementation enters a different state than that specified by the next-state function of the specification. An implementation has a single transfer fault if there is no output fault in the implementation and one and only one of its transitions has a transfer fault. We say that the implementation under test of the given system may have multiple faults if several transitions have output or transfer faults. Here we notice the fault model based on output and transfer faults is often used for diagnosis of a system decomposed into components, where only one component may be faulty [5]. In our context, the specification is a decomposed system; i.e. its implementation is a system of two ComFSMs, where at most one of these machines is faulty.

3. DIAGNOSIS PROBLEM

3.1 Diagnosis Problem Statement

Let $M_1$ and $M_2$ be complete deterministic FSMs representing the specifications of the components of the given system while $M'_1$ and $M'_2$ are their corresponding implementations. We propose an adaptive method for diagnostic test derivation. If the implementation system does not pass a given test suite, our algorithm enables to decide whether it is possible to
identify a faulty component of the given system under the assumption that multiple faults may occur only in one of the implementations $M'_{1}$ or $M'_{2}$. Furthermore, if this is possible, the algorithm enables to decide whether it is possible to locate the faulty transitions within the faulty component, and if possible it locates them. Moreover, the algorithm draws the conclusion “Faults cannot be captured by the assumed fault model” if it has been detected that the implementation at hand has faults that cannot be captured by the assumed fault model.

The diagnostic method can be used for the case where multiple transfer or output faults can occur in one of the implementation component FSMs $M'_{1}$ or $M'_{2}$. However, without loss of generality and for simplicity of presentation, hereafter, we assume that only output faults may occur.

### 3.2 An Overview of the Diagnostic Approach

Let $RS = M_{1} \hat{\diamond} M_{2}$ be a specification system while $TS$ is a conformance test suite. If the composition $M'_{1} \hat{\diamond} M'_{2}$ of the implementations $M'_{1}$ and $M'_{2}$ of the given system produces unexpected output responses to the given test suite $TS$, then the composition $M'_{1} \hat{\diamond} M'_{2}$ is not equivalent to $RS$, i.e. either $M'_{1}$ or $M'_{2}$ is a faulty implementation. Our objective is to determine whether $M'_{1}$ is not equivalent to $M_{1}$ while $M'_{2}$ and $M_{2}$ are equivalent, or vice versa.

In order to determine whether the output responses of $TS$ can be produced when FSM $M'_{2}$ has output faults and $M'_{1}$ is equivalent to its specification, we derive the FSM $(M_{2})_{a}^{out}$ by adding new (faulty) transitions with all possible outputs to each transition of $M_{2}$, and then we combine the obtained non-deterministic FSM with $M_{1}$. We call the obtained non-deterministic machine, $M_{1} \hat{\diamond} (M_{2})_{a}^{out}$, the Fault Function (FF) of the embedded component (or FF-Embedded). Similarly, we derive the FF of the context $M_{1}$, $FF-Context = (M_{1})_{a}^{out} \hat{\diamond} M_{2}$, to determine whether the output responses of $TS$ can be produced when FSM $M'_{1}$ has output faults and $M'_{2}$ is equivalent to its specification. Fault functions were introduced in [8] to represent in a concise way all mutants of a given FSM with a given type of implementation errors.

The set of all deterministic sub-machines of $M_{1} \hat{\diamond} (M_{2})_{a}^{out}$ (or $(M_{1})_{a}^{out} \hat{\diamond} M_{2}$) includes each implementation $M_{1} \hat{\diamond} M'_{2}$ (or $M'_{1} \hat{\diamond} M_{2}$), where output faults are possible in the implementation of component FSM $M_{2}$ (or $M_{1}$). However, the set also includes superfluous sub-machines that do not correspond to a composition of any possible deterministic component machines. This is due to the fact that while deriving the composition, we do
not take into consideration that for a specific state-input combination \((s, i)\), one and only one output is possible in a deterministic implementation.

Since our implementation system is deterministic we do not take into account non-deterministic paths of machines \(M_1 \hat{\circ} (M_2)_a^\text{out}\) and \((M_1)_a^\text{out} \hat{\circ} M_2\). Moreover, we also remove from \(M_1 \hat{\circ} (M_2)_a^\text{out}\) and \((M_1)_a^\text{out} \hat{\circ} M_2\) a behavior that does not agree with the observed outputs to the applied test suite \(TS\). In Section 4, we describe the algorithm that removes from machine \(M_1 \hat{\circ} (M_2)_a^\text{out}\) (or \((M_1)_a^\text{out} \hat{\circ} M_2\)) sub-machines whose output responses to the test suite do not agree with those obtained by applying the test suite \(TS\) to the SUT.

If the SUT is equivalent to a deterministic sub-machine of \(M_1 \hat{\circ} (M_2)_a^\text{out}\) then the faulty machine is \(M_2\), and if it is equivalent to a deterministic sub-machine of \((M_1)_a^\text{out} \hat{\circ} M_2\), then the faulty machine is \(M_1\). However, if the SUT is equivalent to a deterministic sub-machine of \(M_1 \hat{\circ} (M_2)_a^\text{out}\) and to a deterministic sub-machine of \((M_1)_a^\text{out} \hat{\circ} M_2\), then the faulty machine cannot be identified. This is due to the fact that there are some possible faults in \(M_1\) and some possible faults in \(M_2\) that cause the same observable behavior of the SUT. If none of the above cases applies, we conclude that the implementation has faults that are not captured by the considered fault model.

In order to draw one of the above conclusions, we should have test cases such that by observing the output responses of the SUT to these test cases, we can distinguish the SUT and sub-machines of \((M_1)_a^\text{out} \hat{\circ} M_2\) and of \(M_1 \hat{\circ} (M_2)_a^\text{out}\). If the machines \((M_1)_a^\text{out} \hat{\circ} M_2\) and \(M_1 \hat{\circ} (M_2)_a^\text{out}\) have no equivalent deterministic sub-machines then there exists a so-called *distinguishing set* of input sequences [6] such that given the set of deterministic output responses to these input sequences, we always can determine whether the machine under test is a sub-machine of \((M_1)_a^\text{out} \hat{\circ} M_2\) or \(M_1 \hat{\circ} (M_2)_a^\text{out}\). The algorithm for deriving such a distinguishing set is proposed in [6]. We illustrate this algorithm in Section 4. In other cases, it may happen that there are sub-machines of \((M_1)_a^\text{out} \hat{\circ} M_2\) and others of \((M_1)_a^\text{out} \hat{\circ} M_2\) that are equivalent, but these sub-machines are not equivalent to the SUT. In this case, the observed outputs to the given test suite are insufficient to distinguish between these sub-machines and the SUT. Therefore, more test cases must be generated and added to the test suite. This can be done by breaking *FF-Embedded* and *FF-Context* into sub-machines, and by comparing each pair of the obtained sub-machines. Each time when two obtained sub-machines become distinguishable, their distinguishing test is added to the test suite. This enables the elimination of all sub-machines whose output responses to a distinguishing test are different from those observed by applying this test to the SUT. We break the machines by fixing
some transitions as deterministic transitions, i.e. by reducing the non-determinism of Fault Functions. In the worst case, we can come up with explicit enumeration of all faulty machines. However, as the considered examples show, we usually need to fix only a small number of transitions in order to draw a conclusion.

The details of the diagnostic method are presented in Section 5. Meanwhile, in the following section we present its constituent algorithms illustrated by a working example.

3.3 Working Example

Consider the two machines $M_1$ and $M_2$ given in Fig. 1, and their corresponding reference system $RS = M_1 \hat{\otimes} M_2$ shown in Fig. 2. A given complete test suite $TS$ derived from $RS$ is $TS = \{x_1x_1x_1, x_1x_2x_2, x_2x_1x_1x_2, x_2x_2x_2, x_2x_2x_1\}$. $TS$ is derived using the method presented in [1], and it detects any complete FSM $M'_1 \hat{\otimes} M'_2$ that is not equivalent to $RS$, under the assumption that FSMs $M'_1$ and $M'_2$ have up to two states. The set of expected output responses to the $TS$ is as follows:

$$\lambda(s_0, x_1x_1x_1) = y_1y_2y_2; \quad \lambda(s_0, x_1x_2x_2) = y_1y_2y_1; \quad \lambda(s_0, x_2x_1x_1x_2) = y_1y_1y_1y_3; \quad \lambda(s_0, x_2x_2x_1) = y_1y_3y_2.$$

Let us assume that the composition $M'_1 \hat{\otimes} M'_2$ of the implementation component FSMs $M'_1$ and $M'_2$ produces unexpected output responses $y_1y_3y_3$ to $x_1x_1x_1$; $y_1y_1y_3y_2$ to $x_2x_1x_1x_2$ and $y_1y_3y_3$ to $x_2x_2x_1$. (1)

Thus, the composition $M'_1 \hat{\otimes} M'_2$ is not equivalent to $RS$, i.e. either $M'_1$ or $M'_2$ is a faulty component implementation.

Figure 3 includes the FF - Embedded machine, $M_1 \hat{\otimes} (M_2)_{a \text{ out}}$, and the FF - Context $(M_1)_a \hat{\otimes} M_2$, obtained as described in Section 3.2. In this example, the FF - Context is derived under an assumption that in the faulty context implementation, external outputs can only be replaced with external outputs and internal outputs can only be replaced with internal outputs, respectively. This is done in order to have a simple and more readable working example.

Figure 3. (a) Fault function of Embedded Component (b) Fault Function of Context
4 DISTINGUISHING NON-DETERMINISTIC FSMS

In this section, we present the different algorithms of the test suite derivation method, given in the subsequent section, for locating a faulty component FSM.

4.1 Removing Sub-machines of a Non-Deterministic FSM

The following algorithm is used to remove from $(M_1)_{a\text{out}} \hat{\circ} M_2$ (and from $M_1 \hat{\circ} (M_2)_{a\text{out}}$) sub-machines whose output responses to the test cases of TS disagree with those obtained by applying these test cases to the SUT.

Given a complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$, the algorithm returns a smallest sub-machine $A_r$ of $A$ which has the property that each sub-machine of $A$ that comprises $V$ as a subset of its traces is a sub-machine of $A_r$. We note that in our case, the input parts of the sequences of the initial set $V$ are those of the given test suite $TS$, and the output parts are those observed by applying $TS$ to the SUT.

Algorithm 4.1. Removing from FSM $A$ sub-machines that do not match $V$

**Input:** A complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$

**Output:** The smallest subFSM $A_r = (S, I, O, h, s_0)$ of $A$ that contains each sub-machine of $A$, such that the set of its traces comprises $V$, if exists.

**Step-1.** Given FSM $A_1 = A$ and the set $V$ of sequences over alphabet $(IO)^*$, we derive the tree $Tree_1$ of all deterministic paths through $A_1$ labeled with sequences of $V$. Assign $i := 1$ and go-to Step-2.

**Step-2.** If there exists a sequence in $V$ such that no path in $Tree_i$ is labeled with this sequence, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that sub-machine ($A_r$ does not exist, end of Algorithm 4.1). Otherwise, we build a machine $A_{i+1}$ which is a sub-machine of $A_i$ as follows. For each observably reachable node in $Tree_i$, we copy into the corresponding state in $A_{i+1}$ the outgoing transitions from this node. If there are several observably reachable nodes in $Tree_i$ with the same label, we copy for the corresponding state in $A_{i+1}$ only the matching transitions at these nodes (same input/output/next-state values). If there are no matching transitions in $Tree_i$, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that sub-machine ($A_r$ does not exist, end of Algorithm 4.1). For each state in $A_i$, that labels only non-observably reachable nodes in $Tree_i$, we copy all the outgoing transitions from machine $A_i$ into $A_{i+1}$. If no
transition in the machine $A_i$ has been changed (i.e. $A_{i+1} = A_i$), then we have $A' = A_i$ (End of Algorithm 4.1). Otherwise, go-to Step-3.

**Step-3.** At this step we trim $Tree_i$ in order to obtain $Tree_{i+1}$ using the machine $A_{i+1}$ obtained by Step-2 above. For each path in $Tree_i$ that has a node where the output and/or next node of the transition do not match machine $A_{i+1}$, remove this transition and its sub-tree. If all outgoing transitions from some node for an appropriate input have been removed, we remove the incoming transition to this node. If all transitions from the root node for an appropriate input have been removed, then there is no sub-machine in $A$ such that $V$ is a subset of the set of traces of that submachine ($A'$ does not exist, end of Algorithm 4.1). If the trees $Tree_i$ and $Tree_{i+1}$ coincide, then the machine $A' = A_{i+1}$ is derived (End of Algorithm 4.1). Otherwise, increment $i$ by 1 and go back to Step-2.

As an example, we consider $Tree_1$ in Fig. 4-a generated for the fault function of the context, i.e. $A_1 = (M_1)_{a^\text{out}} \bowtie M_2$ in Fig. 3-b, using Step-1 and $TS$. We do not include in $Tree_1$ the paths that do not match the observed output of $TS$. For example, for all test cases in the $TS$ that start with the input $x_2$, the observed output for $x_2$ is $y_1$. Therefore, the paths of $(M_1)_{a^\text{out}} \bowtie M_2$ that start from the initial state $a_1$ by a transition labeled with a label other than $x_2/y_1$ are not included in $Tree_1$. Moreover, we do not include in $Tree_1$ any non-deterministic path. For example, we do not include the path $a_1 \overset{x_1/y_1}{\rightarrow} b_1 \overset{x_1/y_3}{\rightarrow} b_1 \overset{x_1/y_3}{\rightarrow} a_1$.

In $Tree_1$, the root node $a_1$ is observably reachable through the empty sequence. Therefore, in Step-2, we copy the outgoing transitions of that node into $A_2$ (shown in Fig. 4-b), i.e. transitions, $a_1 \overset{x_1/y_1}{\rightarrow} b_1$, $a_1 \overset{x_2/y_1}{\rightarrow} a_2$ and $a_1 \overset{x_2/y_1}{\rightarrow} b_1$. Moreover, in $Tree_1$, starting from the root node $a_1$, the node $b_1$ is observably reachable through the sequence $x_1/y_1$. Therefore, in Step-2, we copy the outgoing transitions of $b_1$ from $Tree_1$ into $A_2$. Nodes $a_2$ and $b_2$ in $Tree_1$ are only non-observably reachable. Therefore, we copy from $A_1$ in Fig. 3-b the outgoing transitions from states $a_2$ and $b_2$ into $A_2$.

Afterwards, using $A_2$ in Step-3, we consider in $Tree_1$, starting from the root node $a_1$, the outgoing transitions from the node $b_1$ that is reached through the sequence $x_2/y_1$. We notice that all its outgoing transitions do not match $A_2$. Therefore we trim the sub-tree of this node, and since all outgoing transitions for inputs $x_1$ and $x_2$ from this node are removed, we remove its
incoming edge, and we get $Tree_2$, which is equal to $Tree_1$ except that the shaded area TRIM-1 is removed.

Back to Step-2, by considering $Tree_2$, the root node $a_1$ is observably reachable through the empty sequence. Moreover, starting from the root node $a_1$, the ending nodes $a_1, b_1,$ and $a_2$ are observably reachable through the sequences $x_1/y_1x_2/y_2, x_1/y_1,$ and $x_2/y_1$, respectively, and node $b_2$ is observably reachable through the sequence $x_2/y_1x_1/y_1$. Therefore, for these nodes, we copy their matching outgoing transitions from $Tree_2$ into $A_3$ in Fig.5-a.

Using the same reasoning we trim from $Tree_2$ the shaded areas of TRIM-3, TRIM-4, and TRIM-5 and we obtain $Tree_3$.

Back to Step-2, by considering $Tree_3$, we notice that all nodes are observably reachable. We copy all the matching outgoing transitions of these nodes and obtain the final machine $A_4$ shown in Fig. 5-b.

We apply Algorithm 4.1. to the $FF$-Embedded, i.e. for $M_1 \diamond (M_2)_{a_{out}}$ of Fig. 3-a, and we obtain the machine shown in Fig. 5-c.

**Theorem 4.1.** Given a complete non-deterministic FSM $A = (S, I, O, h, s_0)$ and a set $V$ of deterministic sequences over alphabet $(IO)^*$, if there exist a sub-machine $B$ of $A$ that has $V$ as a subset of its traces, then the FSM $A^r$ is
derived by Algorithm 4.1 and it includes $B$. Otherwise, the FSM $A'$ does not exist.

**Proof:** In Step-2 of Algorithm 4.1, if there exists no deterministic path in $Tree_i$ labeled with some sequence $\alpha/\beta \in V$, then there is no deterministic sub-machine of $A$ that can produce the observed output sequence $\beta$ to the input sequence $\alpha$, i.e. there is no sub-machine $A'$ in $A$ such that $V$ is a subset of traces of $A'$. Moreover, In Step-3, if all the transitions from the root node in $Tree_i$ have been removed, then there is no deterministic sub-machine $A'$ in $A$ such that $V$ is subset of traces of $A'$. The transitions of $A$ are changed only during Step-2 for the states that label observably reachable nodes of the corresponding tree. For this reason, it is enough to show that each sub-machine $B$ of $A_i$ that has $V$ as a subset of its traces, is a sub-machine of $A_{i+1}$. Let state $s$ label an observably reachable node of $Tree_i$ that is reachable through some sequence $\alpha/\beta \in V$. Any sub-machine of $A_i$ that has $V$ as a subset of its traces must reach state $s$ after applying the sequence $\alpha$. Therefore, all transitions from the state $s$ of $B$ match transitions from this node, i.e. $B$ is a sub-machine of $A_{i+1}$.

### 4.2 Distinguishing the Sets of Deterministic Sub-Machines of Two Non-Deterministic FSMs

If the two sub-machines of the FSMs $FF$-$Context$ and $FF$-$Embedded$ obtained after using the above procedure for removing behaviors that do not match observed outputs, do not have equivalent sub-machines then we can derive a distinguishing set $DisSet$ that allows us to recognize which of the component FSMs is faulty. This set can be constructed using the algorithm given in [6]. In other words, let machines $M_1 \hat{\circ} (M_2)_{a_{out}}$ and $(M_1)_{a_{out}} \hat{\circ} M_2$ be distinguished with the distinguishing set $DisSet$. Let also $P$ be a sub-machine of $M_1 \hat{\circ} (M_2)_{a_{out}}$ or of $(M_1)_{a_{out}} \hat{\circ} M_2$. Then by observing the output responses of $P$ to sequences in the set $DisSet$, we can always conclude whether $P \in Sub(M_1 \hat{\circ} (M_2)_{a_{out}})$ or $P \in Sub((M_1)_{a_{out}} \hat{\circ} M_2)$, i.e. the diagnostic problem is always solvable. Therefore, we derive, using the algorithm given in [6], a distinguishing set $DisSet$ for the FSMs $FF$-$Context$ and $FF$-$Embedded$ in order to recognize a sub-machine that corresponds to the faulty SUT.

As an example, we consider the machines $FF$-$Context$ (machine $A_4$ of Fig. 5-b) and $FF$-$Embedded$ (Fig. 5-c) obtained after deleting sub-machines with traces that do not match observed output responses. The sequence $x_2x_1x_1x_2x_2$ distinguishes these machines. Therefore, if we apply the input $x_2$ after the input sequence $x_2x_1x_1x_2$ and the implementation at hand produces the output response $y_1$ to the tail input $x_2$, then we conclude that $M'_1$ is the
faulty implementation. If the output $y_3$ is produced to the tail input $x_2$, then we conclude that $M'_2$ is faulty. If the implementation produces the output different from $y_3$ and $y_1$, then the implementation at hand has faults that cannot be captured by the assumed fault model.

4.3 Determining a Superfluous Sub-machine

Due to the considered fault model, the SUT $M$ is a sub-machine of $M_1 \circ (M_2)_a^{out}$ or $(M_1)_a^{out} \circ M_2$. However, we mentioned above that not each sub-machine of $M_1 \circ (M_2)_a^{out}$ (or $(M_1)_a^{out} \circ M_2$) can be obtained through output faults in the implementation of $M_2$ (or $M_1$). The reason is that we do not take into account deterministic and non-deterministic paths when combining the compact representation $(M_2)_a^{out}$ of all possible implementations of $M_2$ with $M_1$ (or the compact representation $(M_1)_a^{out}$ of all possible implementations of $M_1$ with $M_2$), and thus we may obtain superfluous sub-machines. Therefore, it may happen that the SUT $M$ is equivalent to a sub-machine of $M_1 \circ (M_2)_a^{out}$ and to a submachine of $(M_1)_a^{out} \circ M_2$, but there is no sub-machine $M'_2 \in (M_2)_a^{out}$ such that $M_1 \circ M'_2$ is equivalent to $M$. In this case, only the implementation of $M_1$ is faulty. Thus, we have the following problem.

Given FSM $M_1 \circ (M_2)_a^{out}$ and its sub-machine $M$, we must check whether there exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M=M_1 \circ M'_2$. To solve the problem we can project sub-machine $M$ onto the set of states of $M_2$ and input and output alphabets of $M_2$. There exists FSM $M'_2 \in (M_2)_a^{out}$ such that $M=M_1 \circ M'_2$ if and only if the obtained FSM is deterministic. A sub-machine $M'$ of $M_1 \circ (M_2)_a^{out}$ for which there is no $M'_2 \in (M_2)_a^{out}$ such that FSMs $M_1 \circ M'_2$ and $M$ are equivalent, is called a superfluous sub-machine.

5. FAULT DIAGNOSIS ALGORITHM

**Algorithm 5.1. Recognizing a faulty component FSM**

**Input:** Composition $M \equiv M_1 \circ M_2$ of two FSMs $M_1$ and $M_2$, and the set $V=TS$ of sequences over alphabet $(IO)^*$.

**Output:** Verdict “Component FSM $M_1$ (or $M_2$) is faulty”, or verdict “Both $M_1/M_2$ could be faulty” when there is a possible faulty implementation of $M_1$ and a possible faulty implementation of $M_2$ that cause the same observable behavior as the implementation at hand, or
verdict “Faults cannot be captured by the assumed fault model” if it has been detected that the implementation at hand has faults that can not be captured by the assumed fault model.

**Step-1.** Derive machines, \( A_1 = (M_1)_a^{\text{out}} \blacklozenge M_2 \) (Fault Function of \( M_1 \)) and \( A_2 = M_1 \blacklozenge (M_2)_a^{\text{out}} \) (Fault Function of \( M_2 \)). Let the set \( R_1 \) be equal to \( \{A_1\} \), and the set \( R_2 \) be equal to \( \{A_2\} \).

**Step-2.** For each machine say \( A_k \) in the sets \( R_1 \) and \( R_2 \), call Algorithm 4.1. to obtain the smallest sub-machine \( A' \) of \( A_k \) which includes all sub-machines of \( A_k \) that have \( V \) as a subset of their traces. If such an \( A' \) exist, replace \( A_k \) by \( A' \). Otherwise, remove \( A_k \) from the corresponding set \( R_1 \) or \( R_2 \).

If the sets \( R_1 \) and \( R_2 \) are empty, then the implementation at hand has a fault that is not captured by the assumed fault model (End of diagnosis algorithm). If the set \( R_1 \) (or \( R_2 \)) is empty, then we conclude the other machine \( M_2 \) (or \( M_1 \)) is faulty (End of diagnosis algorithm).

Otherwise, check, as described in [6], whether there are two machines, say \( A_i \) in \( R_1 \) and \( A_j \) in \( R_2 \), that are distinguishable.

- If there exist such two machines, we obtain, using the algorithm given in [6], the \( \text{Dist}_{\text{set}} \) that distinguish them, and we apply the input sequences of this set to the SUT.
  --If \( |R_1| = 1 \) and \( |R_2| = 1 \), i.e. \( R_1 = \{A_i\} \) and \( R_2 = \{A_j\} \), then:
    -If the output responses of the SUT to the \( \text{Dist}_{\text{set}} \) are different from those expected by both machines \( A_i \) and \( A_j \), then we conclude that the implementation at hand has faults that cannot be captured by the assumed fault model (End of diagnosis algorithm).
    -Else, if the output responses of the SUT are different than those expected by \( A_j \) (or \( A_i \)), then we conclude that \( M_1 \) (or \( M_2 \)) is the faulty machine (End of diagnosis Algorithm).
  --If \( |R_1| > 1 \) or \( |R_2| > 1 \), then after observing the output responses of the SUT to the sequences in \( \text{Dist}_{\text{set}} \), we remove \( A_i \) (or \( A_j \)) from the set \( R_1 \) (or \( R_2 \)) if these responses are different than those expected by \( A_i \) (or \( A_j \)). Then, we add the sequences in \( \text{Dist}_{\text{set}} \) with the observed output responses to \( V \), and we return return back to Step-2.

- If all the machines in \( R_1 \) are indistinguishable from those in \( R_2 \), then
  --If \( R_1 \) and \( R_2 \) have only deterministic machines, then check whether all the machines in \( R_1 \) and in \( R_2 \) are superfluous as described above.
    -If all the sub-machines in \( R_1 \) (or in \( R_2 \)) are superfluous, then machine \( M_2 \) (or \( M_1 \)) is faulty (End of diagnosis algorithm).
-Else, if there exist a sub-machine in $R_1$ and another in $R_2$ that are not superfluous, then we conclude that “Both $M_1/M_2$ could be faulty”. There is a possible faulty implementation of $M_1$ and a possible faulty implementation of $M_2$ that cause the same observable behavior as that of the implementation at hand

--Else, if the set $R_1$ (or $R_2$) has at least one non-deterministic FSM, then we break that machine into $k$ machines by fixing one of its non-deterministic transitions. Then, we replace that machine in the set $R_1$ (or $R_2$) with the obtained sub-machines, and we go back to Step-2.

As another example, suppose that the implementation of the composed system $M'\hat{\circ}M''$ of specifications in Fig. 1 produces the unexpected output responses $y_1y_1y_2y_2$ to the input sequence $x_2x_1x_1x_2$ of TS. This can happen if $M'$ has an output fault; namely it produces $y_2$ instead of $u_2$ on executing transition $t_5$.

By applying the diagnostic method described above, we find that $M'$ can not be identified as the faulty implementation since there exists a faulty implementation $M''$ of $M_2$ such that $M_1\hat{\circ}M''$ and $M'_1\hat{\circ}M_2$ are equivalent. It is the case when $M''$ produces $z_2$ instead of $z_1$ on executing transition $t'_4$.

6. FURTHER RESEARCH WORK

In this paper we presented a method for diagnostic test derivation when the system specification and implementation are given in the form of two communicating finite state machines and at most a single component machine can be faulty. The algorithm can be extended for locating the faults within the faulty machine (if possible). The idea here is to locate the faulty implementation that corresponds to the faulty machine. This implementation is a sub-machine of $(M_1)_{\hat{\circ}M_2}^\text{out}$ when $M_1$ has a faulty implementation, or a sub-machine of $M_1\hat{\circ}(M_2)_{\hat{\circ}M_1}^\text{out}$ when $M_2$ has a faulty implementation. Therefore, the algorithm can be extended to decide which sub-machine of $(M_1)_{\hat{\circ}M_2}^\text{out}$ or $M_1\hat{\circ}(M_2)_{\hat{\circ}M_1}^\text{out}$ is equivalent to the system at hand, i.e. comparison between sub-machines of a given FSM can be added to the algorithm. We note that sometimes it is not possible to locate the faulty sub-machine. It is the case when the SUT has the same observable behavior for different output faults of the faulty machine. Moreover, the algorithm can be extended for locating faults when the specification and implementation systems are given in the form of a single FSM.
Currently, we are performing experiments to estimate the effectiveness and the complexity of the method. The preliminary results obtained for the case when only single output faults in a component are taken into account, show that almost always the faulty component can be identified.

REFERENCES

FROM ACTIVE TO PASSIVE:
PROGRESS IN TESTING OF INTERNET
ROUTING PROTOCOLS

Jianping Wu, Yixin Zhao and Xia Yin
Department of Computer Science, Tsinghua University, Beijing 100084
jianping@cernet.edu.cn, {zhaoyx,yxia}@csnet1.cs.tsinghua.edu.cn

Abstract  Routing protocols play an important role in the Internet and the test
requirements are mounting. To test routing protocols more efficiently, several
enhanced techniques are applied in our Protocol Integrated Test System. The
Implementation Under Test is modeled as a black box with windows. The test
system is endowed with multiple channels and multiple ports to test distributed
protocols. The test suite and other related aspects are also extended.
Meanwhile, a passive testing is introduced to test, analyze and manage routing
protocols in production field, capable of performing the conformance test, the
interoperability test and the performance test. Necessary algorithms and
techniques were worked out for the passive testing to cover the scope from the
protocol state machine to the routing information manipulation. This paper
presents our efforts to test Internet routing protocols with both the active
testing and the passive testing.

Keywords:  routing protocol, test structure, passive testing

1.  INTRODUCTION

High performance and stable routing protocols dominate the performance
of the Internet. Currently the most commonly used routing protocols are RIP
[1], OSPF [2] and BGP [3]. There are other routing protocols like [4-6], but
despite their importance, the test activity is still rather limited due to the fact
that the original test theories, structures and systems were aimed at
communication protocols. While routing protocols possess different features.

This work has been supported by National Natural Science Foundation of China under Grant
No. 69682002 and No.69725003
Several deficiencies were encountered when we started our work on routing protocol test with the Protocol Integrated Test System – PITS [7].

Then the Implementation Under Test (IUT or module under test MUT) is modeled as a windowed black box and tested with a multiple channel test method. The test suite has evolved into a heterogeneous one, while the test data of different types can be described in their appropriate formats. Enhanced techniques mainly focus on the Reference Implementation (RI). In cooperation with the Enhanced RI (ERI), the packet extension technique helps to describe the whole test activity integrally and uniformly.

Some abnormalities will only appear in practice and/or over a long time, such as route oscillations [8], useless route advertisements [9] or even exhausted routers. Although the active testing helps a lot to uncover the deficiencies of the implementations of routing protocols [10] [11], it can hardly deal with these problems. The passive testing [12] can be performed in production field over a long period of time without interference on the network. It is able to perform the conformance, interoperability and performance tests. Therefore, the passive testing has also been adopted in our work on routing protocol test. Corresponding techniques have also been worked out [13][16].

All the following methods are applied in the test activity. We have built an Enhanced Reference Implementation [19] for PITS An intelligent online BGP-4 analyzer[20] has been implemented and then a prototype of a more general online test system [21]. Another important advance is that the active testing and the passive testing are combined, which helps us in the effort to cover all features of routing protocols in our test. These techniques are applicable for RIP, OSPF and BGP. In this paper, we will mainly focus on BGP-4 as the example.

2. FEATURES AND TEST REQUIREMENTS OF ROUTING PROTOCOLS

Compared with communication protocols such as TCP/IP, routing protocols are quite different. First, the purpose of the routing protocol is to collect the information needed to do routing [1], which is a basic summary of routing protocol functions. Another extraordinary character is the nature of distributed systems, which can be found from both the interaction between different routers and the communication between the interfaces of one single router. No matter the designation or the implementation, there are several important databases to store the routing information. The information flow among these databases is connected by the internal processes which dominate the correctness and efficiency of routing protocol implementations.
For a routing protocol, there is no “upper level protocol” at all. However the interaction of routing protocols is much more complex. For example, BGP-4 will talk with neighbor peers while exchanging information with other routing protocols such as OSPF on the same router. A more detailed discussion can be found in [14]. For the testing of communication protocols, the protocol state machine is heavily relied upon. The state machines only describe a small part of the behavior of routing protocols. The majority of a routing protocol activities begins after it enters a “stable” state (e.g. established state for BGP-4).

Therefore, the testing of routing protocol should cover (1) the packet structure, (2) the protocol state machine, (3) the interoperability, (4) exchange and manipulation of routing information, (5) performance evaluation and (6) other features such as the policy-based routing and the route flap damping.

3. CURRENT TEST SYSTEM AND DEFICIENCIES

3.1 Structure of PITS

PITS is the second generation of the test system we developed. Figure 1 shows the structure of PITS. PITS exploits test suites in TTCN[15] format and adopts a concurrent interpretive test execution method for high efficiency. The test suite generation tool TUGEN generates a TTCN-formatted test suite from the formal description of the protocol. Test Execution (TE) is the key engine in the test process, responsible for test process control and the generation of all the information needed in the test report. PITS is a general tester, that is, TE works independently of any particular protocol. All the protocol-related information is reflected in the test suite and the corresponding RI.

![Figure 1 Protocol Integrated Test System](image-url)
3.2 Deficiency and limitation of current test system

Since PITS was mainly aimed at the testing of communication protocols, some deficiencies were uncovered when testing routing protocols.

Originally, RI works in a very simple way, communicating with TE to get commands and data and sending/receiving data packets through the service provider. Test data and correct results are written in the test suite in advance and it is up to TE to accomplish the final comparison and judgement. Since there are large amounts of data transmission and routing information processing and TE depends on interpretive means to execute, TE can hardly accomplish the numerical calculation.

In traditional work, the test system only sets up a communication channel with the MUT and always exploits only one port. With such a test structure, only the packet structure and the protocol state machine can be tested. Since the routing protocol possesses features of distributed systems, the test system must be able to provide multiple ports. Only then can routing information exchange and processes be tested. Even if multiple ports are available, the entire scope of the testability is restricted to the data that the routing protocol module sends out. In addition, testing the correctness of the routing protocol under different conditions is also very important, especially for policy-based routing. Manual configurations are not only time-consuming but also error prone. On the other hand, the routing protocol is always implemented in the router while interacting with other modules. These modules provide the information and the access to the routing protocol module and the whole router through standard interfaces. In the original test structure, since the test system only connects with the MUT, this information and access can’t be exploited at all.

There are further limitations regarding TTCN. It is difficult to describe certain cases with TTCN and it is almost impossible for TE to execute it. We believe that the latest TTCN-3 has greater ability and flexibility. However, the necessary editor, the executing environment, is under development.

The router always has a powerful processing ability while the speed of the test system is relatively much slower. When the test case shown in figure 2 is executed, there may be possible packet loss.

<table>
<thead>
<tr>
<th>No</th>
<th>Behavior Description</th>
<th>Label</th>
<th>Constraints Ref</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1 OPEN PACKET</td>
<td></td>
<td>C1 OPEN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>START T_HOLD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T? KEEPALIVE_PACKET</td>
<td></td>
<td>C1 KEEP</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>? TIMEOUT</td>
<td></td>
<td></td>
<td>FAIL</td>
</tr>
<tr>
<td>6</td>
<td>T? OTHERWISE</td>
<td></td>
<td></td>
<td>FAIL</td>
</tr>
</tbody>
</table>

*Figure 2 The test case with the possibility of packet loss*
When step 1 is interpreted, the data and the command are sent to RI. RI then sends out the packet OPEN_PACKET. MUT will respond with a KEEPALIVE_PACKET. Since the speed of the router is rather fast, this packet is sent back immediately. At that time, if step 3 hasn’t finished its interpretation RI will not accept the response packet. Further, when step 3 is done and RI is informed to receive some packet, what RI actually receives may be some other packet and step 6 will then be entered. The other scenario is that RI receives nothing at all. When the timer T_HOLD expires, step 5 will be entered.

4. ENHANCING TECHNIQUES

In order to solve these problems discussed above, the test system needs to be enhanced accordingly. After some analysis, almost all the enhancements are related to RI.

4.1 Functional sub-modules introduced into RI

In order to release TE from the heavy burden of the calculation, the dynamic calculation and the data generation sub-modules are introduced (shown in figure 3).

![Figure 3 RI and enhanced RI with functional sub-modules](image)

The test data generation sub-module in the ERI selects the test data or generates new test data with some algorithm according to the demand of TE. The packet composer sub-module forms the data packet and sends them out while the dynamic computation sub-module calculates the expected results. Response from the IUT (or MUT) is first processed by the packet parser sub-module, after which the results are compared with what are expected and finally presented to TE. Through these means, ERI accomplishes the calculation originally taken by TE, which improves the efficiency of the test execution.
4.2 Windowed black box model and multiple channels test method

In order to take full advantage of the information and access provided by the system (for example, a router) where the MUT resides, the windowed black box model and the multiple channel test method are adopted. The windowed black box takes into consideration not only the MUT but also other modules interacting with it. These modules offer standard interfaces and provide the tester with several windows to enhance the observation and the control of the MUT in the routing protocol test. Accordingly, the multiple channel test method is exploited. In traditional work, the test system only sets up a communication channel with the MUT, we call this channel a “basic channel”, as in [10]. The principal difference of the multiple channel test method is that not only does the test system set up a communication channel with the MUT, it also sets up channels with other modules interacting with it (shown in figure 4).

![Figure 4. Multiple channels test method and test structure](image)

The $C_0$ is the basic channel between the test system and the MUT. $M_1, M_2, ..., M_k$ are the $k$ modules (will have been tested in advance) interacting with the MUT. The test system sets up channels $C_1, C_2, ..., C_k$ with these modules. The main idea of the multiple channel test method is to take full advantage of the information available to enhance the observability and the controllability, thus promoting the test efficiency and the coverage. The multiple ports needed to test distributed protocols can be easily implemented by expanding the channels.

4.3 Partition of acceptance sub layer and submission sub layer

In order to avoid possible packet loss in high-speed networks, RI is partitioned into two sub layers: the acceptance sub layer and the submission sub layer. The acceptance sub layer takes charge of receiving packets from the MUT, stores them in a buffer and maintains them. The submission sub layer fetches the packets and provides them to TE according to its demand. The acceptance sub layer makes RI independent of the speed of the lower
network and the packet loss can be avoided by managing a proper interval between test case execution.

### 4.4 Relation of enhanced techniques

The structure of the ERI is shown in figure 5, which also explains the relation of these enhancing techniques. The acceptance and the submission sub layers reside at the bottom of the ERI. All the sub layers above can take advantage of them to make up for the speed gap. In the middle of the ERI structure, are basic and extended channels. Each channel can provide multiple ports in order to test distributed routing protocols. Meanwhile, each channel possesses functional sub-modules such as dynamic calculation and data generation (details can be referred to in figure 3) which can provide more processing ability and flexibility. The topside of the ERI contains the command/data dispensation and collection sub-module, communicating with TE, getting data/command, dispensing it to corresponding channels and returning the result to TE for the final verdict.

![Figure 5 Structure of the Enhanced Reference Implementation](image)

It can be seen that the ERI overcomes the problems discussed above and possesses these features: (1) functionally compatible with the former RI; (2) no effect on the syntax and semantic integrity of TTCN; (3) provides multiple channels to increase the observability and the controllability of the MUT; (4) multiple ports to test distributed routing protocols; (5) reduces the burden of TE and improves the flexibility of PITS. (6) enhanced functionality suitable for the routing protocol test.
4.5 Describing the test activity in the test suite integrally and uniformly

The multiple channel test method also brings new problems. One is how to describe the test activity uniformly and integrally. Therefore, the packet expansion technique is raised. For example, four types of BGP-4 packets are defined in [3] and we defined type 128-143 for the data sent through the configuration channel and 144-161 for the auxiliary information channel. This definition does not contradict current usage of packet types and is unlikely to cause future conflicts. However, there is no actual data in the date field of the packet with special type values. Instead, there is only an index. Corresponding channels will look up the data in their mapping tables. This form of representation is a natural extension of the PIXIT (Protocol Implementation eXtra Information) which can act as an external part of the test suite.

5. APPLIED PASSIVE TESTING IN ROUTING PROTOCOL TEST

5.1 Brief of passive testing

Some abnormalities will only appear in practice and/or over a long time, such as route oscillations [8], useless route advertisements [9] and sometimes even exhausted routers. Although active testing can do a lot to promote the quality of the network equipment, it can hardly deal with such problems. Therefore, we also adopted the passive testing [12] in our work. The passive testing is an online test method as shown in figure 6.

Compared with the active testing, the passive testing is simple in structure, while possessing more feasible and powerful abilities to perform the conformance and the interoperability tests and even analysis of the performance. Additionally, it allows more IUTs and OLTs, enabling distributed test methods.
5.2 Algorithms and methods for passive testing

Algorithms to test the protocol state machine can be found in [13]. As to routing information manipulation, one generic method is topology analysis. The main idea is that correct routing information must describe the network topology correctly. We construct the topology from the routes exchanged and compare it with the actual topology. If differences are found, then there may be some errors in the routing information manipulation. Since we can get the constructed network topology from the OSPF routing information base, we mainly focused on the topology analysis for BGP-4, which can be found in [16]. Other methods like the internal process simulation are also exploited. Different modules are implemented to process the routing information collected online similarly to what the processes in the router will do. The correct scene is calculated and compared with the actual result. For BGP-4, the internal process simulation covers the generic route selection, route selection with policy constraints, and the route flap damping, etc.

5.2.1 Testing of route selection with policy constraint

Route selection with policy constraints is an important feature of BGP-4. These policies directly serve the relations of the Internet Service Providers (ISP) and their customers. Three basic relations are classified (client-provider, provider-provider and customer with backup) and corresponding policies are presented. Policy conformance and violations are defined.

**Definition 1.** Policy conformance Conf = \(<\text{Conf}_i, \text{Conf}_e>\), where \(\text{Conf}_i\) is the import conformance which evaluates the process when receiving routing information. \(\text{Conf}_e\) is the export conformance which evaluates the process when advertising routing information.

Suppose \(\text{PS}_r\) is the series of policies to be enforced on the received routes. \(\text{PS}_r = [P_{r1}, P_{r2}, P_{r3}, \ldots, P_m] \forall R_i \in R_{RCV}, R_i' = P_m(P_{r_{n-1}}(\ldots P_{r1}(R_i))))\), denoted as \(R_i' = \text{PS}_i(R_i)\). \(R_i'\) is route of import conformance: \(R_i'\) \(\text{Conf}_i\) \(\text{PS}_r\).

Suppose \(\text{PS}_s\) is the series of policies to be enforced on the routes to be sent out. \(\text{PS}_s = [P_{s1}, P_{s2}, P_{s3}, \ldots, P_{sm}] \forall R_i \in R_{ACP}, R_i' = P_{sm}(P_{s_{m-1}}(\ldots P_{s1}(S_i))))\), denoted as \(R_i' = \text{PS}_s(R_i)\). \(R_i'\) is route-of export conformance: \(R_i'\) \(\text{Conf}_e\) \(\text{PS}_s\).

Then two basic types of policy violations can be defined; other violations can be made out of these two.

**Definition 2.** Penetration: if \(\exists R_k \in R_{ACP}, \neg (\exists R_l \in R_{RCV}, \text{PS}_i(R_l) = R_k)\), then “penetration” occurs, that is, the route is accepted after the route calculation and selection while it actually should not be.

**Definition 3.** Leak: if \(\exists R_k \in R_{SNT}, \neg (\exists R_l \in R_{ACP}, \text{PS}_i(R_l) = R_k)\), then “leak” occurs, that is, the route is delivered while it actually should not be.
A separate module was implemented. With policy configurations (provided by the network operator), this module can find the basic policy violations and other subtle abnormalities.

5.2.2 Testing of route flap damping

Route flap is another serious problem of the Internet routing [9][17] and the “merit way” is the damping method now in use [18]. However, to our best knowledge, there is no test to evaluate the damping effect. Therefore, we worked out a simple but practical one (shown in figure 7). In this state machine, the flap of a single route, the interaction between two routes (replacement) and the condition with an alternative route are all covered. Since the route selection will finally go down to the comparison between two routes, this model is applicable for generic conditions.

The flap module follows this state machine and performs the merit value calculation (increase and decay). Therefore, not only can the passive testing find the existence of the route flap and its severity, it can also evaluate the effect of the route flap damping.

6. IMPLEMENTATIONS AND TEST ACTIVITIES

6.1 PITS with ERI

The basic channel can support up to 8 ports and two extended channels: the auxiliary information and the configuration channels. The auxiliary information channel is set up to access the information needed in the routing protocol test in its session with the SNMP; the configuration channel connects through telnet and configures the routing protocol and the router with corresponding commands. A test case is shown in figure 8.
Dynamic Part

<table>
<thead>
<tr>
<th>No</th>
<th>BehaviorDescription</th>
<th>Label</th>
<th>Constraints</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T! TO_CONF</td>
<td>C_TO_CONF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ESTABLISHED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T! UPDATE01</td>
<td>C_UPDATE01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T! UPDATE01</td>
<td>C2_UPDATE01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>START T_HOLD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T? FROM_AUX</td>
<td>C_FROM_AUX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>STARTT KEEP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>T?KEEP_ALIVE01</td>
<td>C_KEEPALIVE01</td>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>? TIMEOUT</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>T? OTHERWISE</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>? TIMEOUT</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>T? OTHERWISE</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Comments: NULL

In this case, the configuration channel and the auxiliary information channel are exploited. This example also demonstrates how the test activity is described integrally and uniformly.

In the acceptance sub layer, a buffer of 1600 X 40 bytes is set aside. There is no packet loss in a 10M or 100M Ethernet environment.

6.2 Online intelligent analyzer

The intelligent online BGP-4 Analyzer [20] is our first application of the passive testing. Compared with earlier work, this analyzer offers more efficient means for both the protocol test and the network management. It can test the state machines of BGP peers’, from which some conclusions on the conformance and the interoperability can be drawn. As to the routing information manipulation, the analyzer simulates internal processes, such as the route advertisement and route withdrawal, the policy monitoring and the flap damping. The analyzer can be set to focus on some specified routing information or on certain BGP speakers’ behavior. It can also collect statistics and provide basic performance parameters regarding the frequency and the amount of the data interaction. Also, the analyzer can re-display the procedure, making it easy to analyze and eliminate faults. Further, it can generate a TTCN-formatted test suite with the recorded data. Besides, it has a friendly graphical user interface (GUI) running on Linux and can attend to multiple NICs simultaneously. Data is stored in a compressed form in consideration of working over a long period time.
6.3 Online test system

As a more generic system for all the routing protocols and other distributed protocols, we have implemented the prototype of an online test system [21]. The framework is shown in figure 9. There are five main components. The input component includes a protocol state machine description module, a protocol packet structure description module and a set condition module. The communication component includes a communication interface module and a condition check and filter module, which shields the difference among communication protocols of different layers. The condition check and filter module further processes the data collected and provides the test component with a uniform interface. The test component is the kernel of the OLTS, including the protocol state machine test module, the internal process simulation module and the statistics module. The cooperation component is used to organize multiple OLTS instances. The test result module, the LOG module and other modules belong to the conclusion and presentation component, which composes a test report, provides graphical presentations and generates the test suite.

For each protocol to test, a specific communication interface module is coded, descriptions of the protocol state machine and the protocol packet structure are input and the necessary internal process simulation module is applied or implemented. OLTS is implemented on a Sun Ultra 1 - Solaris and a x86-Linux separately in order to widen its application and verify its portability.

Specific modules will be implemented for different communication media to shield the difference of the lower level. For example, the interface has been implemented for the most popular Ethernet, which is able to capture all the data of upper level protocols. On x86-Linux, the packet socket is exploited and DLPI (Data Link Provider Interface) [22] operating /dev/le on the Sun Ultra 1 - Solaris. For other communication media, there has been no difficulty encountered so far (even for a non-broadcast point to point link).
Based on this, the virtual network interface (VNI) is further implemented. VNI aims to provide a uniform interface that the main process can operate. When new communication media is introduced, the only requirement is to map its interface to the uniform interface.

The internal process simulation is done via a module stack; the modules can be pushed onto the stack for different protocols and test purposes.

In consideration of testing in a distributed environment, Simple Cooperation Protocol (SCP) [23] is defined for the cooperation component, which enables multiple instances of OLTS’s to work together in different places of the network. We still have a tool set, including the packet composer. It can be controlled through a control channel by other systems such as PITS.

One of the most interesting aspects is that the active testing and the passive testing are combined. The active testing and the passive testing have both advantages and limitations of their own. The active testing can send out the desired packets; the passive testing can work in production field over a long time. By combining the active testing and the passive testing, we can expect a better efficiency.

An example is the erroneous session shown in figure 10. It is first captured by the analyzer. 166.111.69.91 is one port address of the commercial router and 166.111.69.75 is that of the Router II. This session shows that due to the incorrect implementation of BGP-4 of Router II, UPDATE packets are sent out repeatedly instead of OPEN after the TCP connection is established. But the response of the commercial router is not correct either. After it responds with a NOTIFICATION packet (line 2 in figure 10), it answers with the same UPDATE (line 5 in figure 10), which contradicts the protocol specification and leads to problems of the conformance and the interoperability along with the extra cost of the network bandwidth.

Then this fault was confirmed with the active testing and several other similar faults were uncovered with this indication.

![Figure 10](image-url) An erroneous session recorded

Figure 10 An erroneous session recorded
The test structure used in the policy-based routing test is shown in figure 11. BG1 is a real router, while W1 and W2 are two online test systems. N1 and N2 are two nodes, each of which may be a real router or an active test system. N1 sets up an EBGP connection with BG1 and N2. BG1 an IBGP connection, W1 watches the session on EBGP and W2 on IBGP. With different configurations, variant tests can be performed. In a production field, N1 and N2 are all real routers, online test systems W1 and W2 monitor if there is any leak or penetration. Under the constructed test environment, N1 and N2 are test systems and they can actively test policy-based routing to detect violations.

The test structure of the route flap damping (shown in figure 12) also covers both the active testing and the passive testing. There are three types of nodes in this structure: flap source nodes, router nodes and detector nodes. The router nodes construct a network. The flap source nodes inject flap into the network at different positions and the detector collects the information about the flap damping. The test is in an active one. However, if the network is a real network, it will generate flap itself. Then the set of flap source nodes can be empty and the test switches into a passive mode.

**Figure 11** Test structure for policy based routing test

**Figure 12** Test structure for route flap damping

**Figure 13** Topology constructed with the data collected from the backbone of CERNET
With both the active testing and the passive testing, a large number of test activities have been performed. We have tested some commercial routers in our lab and we have performed the test on the backbone of CERNET [24]. Figure 13 is the topology constructed with the data collected on CERNET. We compare this constructed topology with the actual one to discover errors in routing information manipulation. We also offer test support to the development of routers. Their pass ratio is considerably promoted and therefore the quality is improved accordingly.

7. CONCLUSIONS

Despite the great importance of routing protocols, test activity is still quite limited compared with the testing of communication protocols. In addition to our previous work, reference [11] discusses the testing of IP routing protocols in the environment generated by software tools. Craig Labovitz has collected the routing data for two years and published several papers [17]. Currently there are some commercial test systems, such as HP Router Tester [25] and NetCom Systems Smartbits [26], etc. They mainly focus on the performance of the system. They are all active testers and possess a high degree of industrialization. The price is rather expensive and they are always difficult to extend. For example, users can’t develop test suites for new protocols.

This paper presents our efforts to cover important features of Internet routing protocols. However, there is still quite a lot to do. Routing protocols are still progressing, while models, methods and other techniques still need further refinement. At the same time, more efforts must be expended within the field of the protocol test, through which the validity of new methods can be proved and new problems can be discovered.

REFERENCES


[22] OSI. Data Link Provider Interface Specification, UNIX International OSI Work Group, revision 2.0.0. 1991


Part Three

Timed Automata
This page intentionally left blank.
TIME AND ACTION LOCK FREEDOM PROPERTIES FOR TIMED AUTOMATA

Howard Bowman
Computing Lab, Univ of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK
H.Bowman@ukc.ac.uk

Abstract  Time and action locks can arise freely in timed automata specification. We review techniques for avoiding the occurrence of timelocks, based on the Timed Automata with Deadlines framework. Then we present a notion of parallel composition which preserves action lock freeness, in the sense that, if any component automaton is action lock free, then the composition will also be action lock free.

Keywords: Timed Automata, Timed Systems, Timelocks.

1. INTRODUCTION

Deadlocks are the characteristic error situation arising in concurrent and distributed systems. In very general terms, they are states in which the system is unable to progress further.

Classically the term deadlock has been seen as synonymous with what we will call action locks. These are situations in which, how ever long time is allowed to progress, the system will never be able to perform an action. Such action locks often result from unmatchable action offers, e.g. when a component wishes to perform a synchronisation action, but is unable to because no other process can offer a matching synchronisation.

In the context of timed systems, new locking situations arise. In particular, in this paper, we will be working in an environment with two main types of locking situation. As a result of this, we have had to be careful with our choice of terminology. Thus, in this paper the term deadlock is the most general. It embraces action locks and the form of locking behaviour that comes with timed systems - timelocks.

Timelocks are situations in which, informally speaking, time is prevented from passing beyond a certain point. They are highly degenerate occurrences [4] because they yield a global blockage of the systems evo-
lution. This is quite different from an action lock, which cannot affect the evolution of an independent process.

In particular, unless significant care is taken, the possible presence of timelocks is a major problem for the formal specification and analysis of time critical networks and distributed systems. This is especially the case when using timed automata, which are at the heart of current timed model checking technology, such as UPPAAL [1] and Kronos [7].

This problem was highlighted in [6] where a number of timelock errors were discovered in a timed automata model of a lip-synchronisation protocol. Furthermore, it was shown in [4] that, when using timed automata, even the simple task of defining a timeout in a communication protocol is hampered by the possible presence of timelocks.

In fact, the issue of whether timelocks are desirable or undesirable features of timed models remains a hotly debated topic. The standard argument in favour of models containing timelocks is that they represent specification inconsistencies (like logical contradictions) and that by discovering and eliminating them, specifications can be corrected. However, we take the contrary position for a number of reasons.

Firstly, detecting timelocks is, in fact, a difficult and expensive analysis task. The classic method is to show that a property such as the Kronos formula, \( \text{init} \Rightarrow \square \diamond = 1(\text{true}) \), holds over the specification. This is an unbounded liveness property which is one of the most difficult classes of formulae to check. Recent work by Tripakis [9] offers potential improvements in such analysis. However, his algorithm remains unimplemented and furthermore, such improvements will always be thwarted by systems with fundamentally large state spaces.

We are also strongly of the opinion that inconsistencies and contradictions fit in the domain of logical description, but are difficult to reconcile with behavioural specification techniques, such as Timed Automata (TA). Contradictions arise when conflicting properties are asserted / conjoined. However, although the mistake is frequently made, parallel composition of processes is not a property composition operator, rather its meaning is operational - two (or more) physical components are run in parallel. Error situations in behavioural techniques should have a behavioural / operational intuition that is justifiable in terms of real world behaviour. This is the case for action locks and livelocks. However, there is no real world counter-part for time stopping!

It is also important to emphasize that the situation with action locks and timelocks is, in this respect, a little different. In our opinion timelocks are highly counter-intuitive and thus we believe that constructively preventing the occurrence of timelocks is essential. However, since action locks are not in the same way counter-intuitive, prevention is not in the
same sense essential. Nonetheless investigating techniques which ensure action lock freeness is useful since it highlights forms of parallel composition that can be employed when building systems that are “correct by construction”.

Although pleasingly simple, as previously implied, timed automata have the weakness that timelocks can freely arise and in a number of different ways. Perhaps most problematically they can arise through the interplay of urgency and synchronous interaction. We argue that urgency is given too strong an interpretation in timed automata, in the sense that an action can be forced (i.e. it becomes urgent) even if it is not possible (i.e. is not enabled).

The first part of this paper presents a re-interpretation of synchronisation that weakens the effect of urgency and thus limits the occurrence of timelocks. The approach uses ideas from the Timed Automata with Deadlines (TADs) framework of Bornot and Sifakis [2, 3]. However, we adapt the TADs definitions in the manner we introduced in [4].

The timed prioritised choice features offered by the TADs framework yield the possibility that the dynamic enabling of “competing” transitions can be defined statically. Hence notions of parallel composition that preserve different dynamic properties can be investigated. In this vein we also present a notion of parallel composition which preserves action lock freeness, in the sense that, if any of the component TADs is action lock free then the parallel composition will also be action lock free. Thus, in this paper we develop notions of parallel composition which are strongly compositional in the sense that if component processes are time and action lock free, then the parallel composition of these process will also be time and action lock free.

Structure of Paper. Section 2 introduces basic notation. Then we tackle the issue of timelocks in section 3 and action locks in section 4. Finally, section 5 presents concluding remarks.

2. BACKGROUND

Notation. $H \mathcal{A} = \{ x? , x! | x \in \mathcal{C} A \}$ is a set of half (or uncompleted) actions. $\mathcal{C} A$ is a set of completed (or internal) actions. These give a simple CCS or UPPAAL [1] style point-to-point communication. Thus, two actions, $x?$ and $x!$ can synchronise and generate a completed action $x$. $A = H \mathcal{A} \cup \mathcal{C} A$ is the set of all actions. We use a complementation notation over elements of $A$: $\overline{x} = x$ if $x \in \mathcal{C} A$, $\overline{x}?$ = $x!$ and $\overline{x}! = x?$.

$R^+$ denotes the positive reals without zero and $R^+0 = R^+ \cup \{0\}$. $\mathcal{C}$ is the set of all clock variables, which take values in $R^{+0}$. $\mathcal{C} \mathcal{C}$ is a set
of clock constraints. Also if \(C \subseteq C\) we write \(CC_C\) for the set of clock constraints generated from clocks in \(C\).

\[ V_C = C \rightarrow R^{+0} \] is the space of clock valuations for clocks in \(C\).

**Timed Automata.** An arbitrary TA has the form: \((L, l_0, T, I, C)\), where, \(L\) is a finite set of locations; \(l_0 \in L\) is a start location; and \(C\) is the set of clocks of the TA.

\(T \subseteq L \times A \times C \times C \times P(C) \times L\) is a transition relation. A typical element of \(T\) would be, \((l_1, e, g, r, l_2)\) where \(l_1, l_2\) are locations; \(e\) labels the transition; \(g\) is a guard; and \(r\) is a reset set. \((l_1, e, g, r, l_2) \in T\) is typically written, \(l_1 \xrightarrow{e,g,T} l_2\), stating that the automaton can evolve from location \(l_1\) to \(l_2\) if the (clock) guard \(g\) holds and in the process action \(e\) will be performed and all the clocks in \(r\) will be set to zero.

\(I : L \rightarrow CC\) is a function which associates an invariant with every location. Intuitively, an automaton can only stay in a state while its invariant is satisfied.

It is important to understand the difference between the role of guards and of invariants. In this respect we can distinguish between *may* and *must* timing. Guards express may behaviour, i.e. they state that a transition is possible or in other words *may* be taken. However, guards cannot "force" transitions to be taken. In contrast, invariants define must behaviour. This must aspect corresponds to urgency, since an alternative expression is that when an invariant expires, outgoing transitions must be taken straightaway.

**Semantics.** TA are semantically interpreted over transition systems which are triples, \((S, s_0, \Rightarrow)\), where, \(S \subseteq L \times V_C\) is a set of states; \(s_0 \in S\) is a start state; and \(\Rightarrow \subseteq S \times Lab \times S\) is a transition relation, where \(Lab = A \cup R^+\). Transitions can be of one of two types: discrete transitions, e.g. \((s_1, e, s_2)\), where \(e \in A\) and time transitions, e.g. \((s_1, d, s_2)\), where \(d \in R^+\) and denotes the passage of \(d\) time units. Transitions are written:

\[ s_1 \xrightarrow{e} s_2 \] respectively \[ s_1 \xrightarrow{d} s_2. \]

For a clock valuation \(v \in V_C\) and a delay \(d\), \(v+d\) is the clock valuation such that \((v+d)(c) = v(c) + d\) for all \(c \in C\). For a reset set \(r\), we use \(r(v)\) to denote the clock valuation \(v'\) such that \(v'(c) = 0\) whenever \(c \in r\) and \(v'(c) = v(c)\) otherwise. \(v_0\) is the clock valuation that assigns all clocks to the value zero.

The semantics of a TA \(A = (L, l_0, T, I, C)\) is a transition system, \((S, s_0, \Rightarrow)\), where \(S = \{ s' \in L \times V_C \mid \exists s \in S, \ y \in Lab . s \xrightarrow{y} s' \} \cup \{ [l_0, v_0]\}, s_0 = [l_0, v_0]\) and \(\Rightarrow\) is defined by the following inference rules:-

\[
\frac{l \xrightarrow{e,g,T} l'}{[l,v] \xrightarrow{e} [l', r(v)]}
\]

\[
\frac{\forall d' \leq d. I(l)(v + d')}{[l,v] \xrightarrow{d} [l, v + d]}
\]
The semantic map which generates transition systems from TA is written \([\mathbb{[}A]\)$. Also, notice that our construction ensures that only reachable states are in \(S\).

**Parallel Composition.** We assume our system is described as a vector of TA, denoted, \(|A = \langle A[1], \ldots, A[n] \rangle|\) where \(A[i]\) is a timed automaton. In addition, we let \(u, u'\) etc, range over the set \(U\) of vectors of locations, which are written, \(<u[1], \ldots, u[n]\>\). We use a substitution notation as follows: \(<u[1], \ldots, u[j], \ldots, u[n]\>[u[j]'=u[j]] = <u[1], \ldots, u[j]', \ldots, u[n]\>\) and we write \([u[j]'=u[j]]\) as \([j'=j]\) and \(u[i'_1/i_1] \cdots u[i'_n/i_n]\) as \(u[i'_1/i_1] \cdots u[i'_n/i_n]\).

If \(\forall i(1 \leq i \leq n) . A[i] = (L_i, l_{i0}, T_i, I_i, C_i)\) then the product automaton, which characterises the behaviour of \(|\langle A[1], \ldots, A[n] \rangle|\) is given by, \((L, l_0, T, I, C)\) where \(L = \{ |u | u \in L_1 \times \ldots \times L_n \}\), \(l_0 = \langle l_{10}, \ldots, l_{1n} \rangle\), \(T\) is as defined by the following two inference rules, \(I|\langle \langle u[1], \ldots, u[n] \rangle\rangle| = I_1(u[1]) \wedge \ldots \wedge I_n(u[n])\) and \(C = C_1 \cup \ldots \cup C_n\).

\[
\begin{align*}
\frac{u[i] = x.g_r u[j] \quad u[j'] \quad x.g_r u[j']}{u[i] = x.g_r u[i']} & \quad x \in CA \\
\frac{u[i] = x,g_r u[i] \quad u[i']}{u[i] = x,g_r u[i']} & \quad x \leq k \neq r \leq y \quad x \leq r \leq y \wedge k \neq r.
\end{align*}
\]

**Timelocks.** We can formulate the notion of a timelock in terms of a testing process. Consider, if we take our system which we denote \(System\) and compose it in parallel with the timed automaton, \(Tester\), shown in figure 1, where, since it is completed, the \(zzz\) action is independent of all actions in \(System\). Then for any \(d \in \mathbb{R}^+\), if the composition \(|<Tester(d), System\rangle|\) can evolve to a state from which it cannot perform \(zzz\), then the system contains a timelock at time \(d\). Notice that we are not saying that all executions of \(System\) will timelock, but rather that \(System\) can timelock.

This illustration indicates why timelocks represent such degenerate situations - even though the \(Tester\) is in all respects independent of the system, e.g. it could be that \(Tester\) is executed on the Moon and \(System\) is executed on Earth without any co-operation, the fact that the system cannot pass time prevents the tester from passing time as well. Thus, *time really does stop* and it stops everywhere because of a degenerate piece of *local* behaviour.

We can also give a semantic definition of the notion, which is similar to definitions given in [9]. However, we first need a little notation. A trace of a timed automaton \(A\) has the form, \(\rho = s_0; y_1; s_1 \ldots s_{n-1}; y_n; s_n\), where, \(\forall i(0 \leq i \leq n) . s_i \in \mathbb{[}A\].1\) (throughout the paper we use the notation \(t.i\) to access the \(i\)th element of a tuple); \(s_0 = [I_0, v_0]\); \(y_i \in A \cup \mathbb{R}^+\); and
∀i(0 ≤ i ≤ n − 1). s_i \xrightarrow{y_i+1}s_{i+1}. We let Tr(A) denote the set of all traces of A and we define the function delay as,

\[ \text{delay}(\rho) = \Sigma\{ y_i \mid 1 \leq i \leq n \land y_i \in \mathbb{R}^+ \} \]

Now we say that A can timelock at time d iff

\[ \exists \rho \in Tr(A) . (\text{delay}(\rho) < d \land \forall \sigma \in Tr(A) . (\rho \text{ pref } \sigma \implies \text{delay}(\sigma) < d)) \]

where \( \rho_1 \text{ pref } \rho_2 \) if and only if \( \rho_1 \) is a prefix of \( \rho_2 \). Intuitively this expresses that there is a state reachable before \( d \) time units has passed, from which it is not possible for time to elapse beyond \( d \). Notice this definition does not preclude the system evolving “while timelocked”.

![Figure 1. A Tester and Timelock Illustrations](image)

There are two different forms of timelock:-

1. **Zeno Timelocks.** These arise when the system has an infinite behaviour but time cannot pass beyond a certain point. In other terms, an infinite number of discrete transitions are performed in a finite period of time. An example of such a specification is System1 (see figure 1).

2. **Time Action Locks.** These are situations in which a state is reached from which neither time or action transitions can be performed. An example of such a lock is a trivial TA with no transitions and one location with invariant false. However, more problematically, time action locks can arise through mismatched synchronisations, e.g. \( \langle \text{System2, System3} \rangle \) (from figure 1) contains a timelock at
time 2, which arises because **System2** must have performed (and thus, synchronised on) action **xxx** by the time t reaches 2 while **System3** does not start offering **xxx** until after t has past 2.

The interesting difference between these two varieties of timelock is that the first one locks time, but it is not action locked. However, the second reaches a state in which neither time passing or action transitions are possible.

A relevant property which appears in the literature is that of time reactivity. It ensures that a system is time action lock free.

**Definition 1** A system is said to be time reactive if it can never reach a state in which neither time nor action transitions can be performed.

**Action Locks.** Timelocks are much more serious faults than action locks, since the latter generate local deadlocks, however, cannot prevent an independent process from evolving. A TA is action locked when it reaches a state from which, however long time is allowed to pass, an action will never be possible. The natural interpretation of action lock in the setting of timed systems is as follows.

**Definition 2** A state \([l, v]\) of a TA \(A\) is an action lock, denoted \(AL([l, v])\), if and only if, \(\forall t \in \mathbb{R}^+ ([l, v + t] \in \big[ A \big].1 \Rightarrow [l, v + t] \not\rightarrow\), where \([l, v + t] \in \big[ A \big].1\) implies \([l, v + t]\) is reachable from \([l, v]\) by the definition of \(\big[ \big]\). A TA \(A\) contains an action lock iff \(\exists s \in \big[ A \big].1 \cdot AL(s)\).

3. **TIMELOCKS**

**Zeno Timelocks.** Using an approach of Tripakis [9] we introduce a static construction which ensures zeno timelock freeness. The idea is to ensure that for each loop in an automaton, time must pass by at least \(\varepsilon \in \mathbb{R}^+\) on every iteration (this is similar to imposing time guardedness in timed process algebra). First a definition.

**Definition 3** For \(A \in TA\) we define a structural loop to be a sequence of distinct transitions, \(l_0 \xrightarrow{\epsilon_1 \cdot g_1} l_1 \xrightarrow{\epsilon_2 \cdot g_2} l_2 \ldots \xrightarrow{\epsilon_n \cdot g_n} l_n\), s.t. \(l_0 = l_n\).

\(A\) is called strongly non-zeno if, for every such structural loop, there exists a clock \(c \in A.5, \varepsilon \in \mathbb{R}^+\) and \(0 \leq i, j \leq n\) s.t., (1) \(c \in r_i\); and (2) \(c\) is bounded from below in step j, i.e. \((c < \varepsilon) \cap g_j = false\).

Clearly, **System1** of figure 1 fails to be strongly non-zeno since a suitable \(\varepsilon \in \mathbb{R}^+\) does not exist. The following result was presented in [9].

**Proposition 1** If \(A \in TA\) is strongly non-zeno then \(Tr(A)\) does not contain a path that is both infinite and yields a timelock.
In addition, strong non-zenoness is well behaved through parallel composition. Specifically, the following result was presented in [9]. It ensures that we cannot generate new zeno timelocks through parallel composition.

**Proposition 2** If $A_1, \ldots, A_n \in TA$ are strongly non-zeno then $\langle A_1, \ldots, A_n \rangle$ is also strongly non-zeno.

Also although we have no empirical evidence, in accordance with [9], we believe that in practice, specifications will invariably be strongly non-zeno.

**The Nature of Synchronisation.** Perhaps the most counter-intuitive aspect of the timelock story is the manner in which timelocks can arise from mis-matched synchronisations, such as the composition in figure 1. If we consider how this problem arises we can see that it is caused by the particular interpretation of urgent interaction employed in TA.

It is without doubt true that facilities to express urgency are required. However, it is our perspective that while urgency is needed, currently it is given an excessively strong formulation. We illustrate the issue with the following example.

**Example.** Consider the specification of the Dying Dining Philosophers problem. The scenario is basically the same as the Dining Philosophers except here we have extra constraints which state that philosophers die if they do not eat within certain time periods.

For example, if at a particular state, Aristotle must eat within 10 time units to avoid death, in TA his situation could be represented as state 10 of timed automaton Aris1 in figure 2. In addition, if say the fork he requires is being used by another philosopher, the relevant global behaviour of the rest of the system might correspond to Rest1 in state m0 (see figure 2 again).

![Figure 2. Dying Dining Philosophers Automata](image)

However, the formulation $\langle \text{Aris1, Rest1} \rangle$ will timelock when $t$ reaches 10. This seems counter-intuitive. Aristotle knows he must pick-up his
fork by a certain time otherwise drastic consequences will result for him (this is why he “registers” his \textbf{pick} request as urgent). However, if he \textit{locally} fails to have his requirement satisfied, he cannot \textit{globally} prevent the rest of the world from progressing, rather a \textit{local} deadlock should result. As a consequence Aristotle might be dead, but as we all know, “the world will go on!”

Conceptually what is happening is that Aristotle is enforcing that his \textbf{pick} action must be taken \textit{even if it is not possible}, i.e. it is not enabled. However, we would argue that urgency can only be forced if an action is possible. In other words, it should only be possible to make an action urgent if it is enabled, i.e.

\begin{quote}
    \textit{must requires may or, in other terms, you can only force what is possible.}
\end{quote}

One way in which such an interpretation of urgency has previously been obtained is through only allowing urgency to be applied to internal actions. This is the so called \textit{as soon as possible} (asap) principle [8], much discussed in the timed process algebra community. This property indeed prevents the occurrence of timelocks due to synchronisation mismatches, but unfortunately, it is not a suitable solution for timed automata. This is because TA do not have a hiding operator. In timed process algebra with asap the hiding operator, which turns observable into internal actions, has an important role since (implicitly) it makes actions urgent. The absence of hiding in TA means that we cannot (selectively) take an observable action that results from synchronising half actions and turn it into an (urgent) internal action.

Thus, now we consider an alternative framework for TA specification - \textit{Timed Automata with Deadlines} (TADs) which was initially devised by Bornot and Sifakis [2, 3] and with which we can obtain the synchronisation interpretation we desire. Our presentation follows that in [4], with some refinements.

\textbf{TADs Basics.} For a full introduction to TADs, we refer the interested reader to [2, 3]; here we highlight the main principles.

\textit{Deadlines on Transitions.} Rather than placing invariants on states, deadlines are associated with transitions. In order to do this, transitions are annotated with 4-tuples, \((e, g, d, r)\), where \(e\) is the transition label; \(g\) is the guard; \(d\) is the deadline; and \(r\) is the reset set. Conceptually, deadlines state when transitions \textit{must} be taken and taken immediately. Since we have deadlines on transitions there is no need for invariants on states. It is also assumed that the constraint, \(d \Rightarrow g\) holds, which ensures that if a transition is forced to happen it is also able to happen. As a result of this constraint, \textit{TADs are time reactive}. 
(Timewise) Priorities. By restricting guards and deadlines in choice
contexts, prioritised choice can be expressed, e.g. if we have two tran-
sitions, \( b_1 = (e_1, g_1, d_1, r_1) \) and \( b_2 = (e_2, g_2, d_2, r_2) \), and we are at a
state with a choice between them, then we can give \( b_2 \) priority over \( b_1 \)
by restricting the guards and deadlines of \( b_1 \) to \( (e_1, g_1', d_1', r_1) \). The
form of priority we use is to enforce the following restricted guard and
deadline, \( g_1' = g_1 \land \Box \neg g_2 \) and \( d_1' = d_1 \land g_1' \), where \( \Box \) is the
temporal operator henceforth. This ensures that \( b_1 \) is only enabled if
\( g_1 \) holds and there is no point in the future at which \( g_2 \) will hold.

Parallel Composition with Escape Transitions. The TADs framework
employs a different parallel composition operator to that arising in stan-
dard timed automata. The key idea is that of an escape transition.
These are the local transitions of automaton components that are com-
bined when generating a synchronisation transition. Thus, not only are
synchronisations included, but component transitions of the synchroni-
sation are as well. The timewise priority mechanism is then used to give
the synchronisation transition highest priority. Intuitively, the escape
transitions can only happen if the synchronisation transition will never
be enabled.

In fact, in addition to ensuring time reactivity, the TADs framework
limits the occurrence of action locks. Specifically, the escape transitions
allow the components of a parallel composition to escape a potential
action lock by evolving locally.

We briefly review the definition of TADs. An arbitrary TAD, has the
form, \( (L, l_0, \to, C) \), where \( L \) is a finite set of locations; \( l_0 \) is the start
location; and \( C \) is the set of clocks.

\[ \to \subseteq L \times A \times C \times C \times \mathbb{P}(C) \times L \]

is a transition relation. A
typical element of which is, \( (l_1, e, g, d, r, l_2) \), where \( l_1, l_2 \in L; e \) la-
bles the transition; \( g \) is a guard; \( d \) is a deadline; and \( r \) is a reset set.

\( (l_1, e, g, d, r, l_2) \in \to \) is typically written, \( l_1 \xrightarrow{e,g,d,r} l_2 \).

As was the case with TA, TADs are semantically interpreted as trans-
ition systems. The following two inference rules are used for this,

\[
\frac{\to \subseteq L \times A \times C \times C \times \mathbb{P}(C) \times L}{(S1)} \quad (S2) \quad \frac{\forall l'. l \xrightarrow{e,g,d,r} l' \quad g(v)}{[l, v] \xrightarrow{e,g,d,r} [l', v(v)]} \quad \frac{\forall l'. l \xrightarrow{e,g,d,r} l' \quad t < t' \iff d(v + t')}{[l, v] \xrightarrow{e,g,d,r} [l, v + t]}
\]

Now we define the semantic map \( \llbracket \cdot \rrbracket \) from TADs to transition systems
as follows, \( \llbracket (L, l_0, \to, C) \rrbracket = (S, s_0, \Rightarrow) \) where, \( S = \{ s' \in L \times V_C \mid \exists s \in S, y \in \text{Lab} \cdot s \xrightarrow{y} s' \} \cup \{[l_0, v_0]\}; s_0 = [l_0, v_0]; \) and \( \Rightarrow \subseteq (L \times V_C) \times \text{Lab} \times (L \times V_C) \) satisfies \( (S1) \) and \( (S2) \). Notice that, once again, \( S \) only
contains reachable states.
In addition, we will use the function:

\[ \theta_B(l) = \{ (e, g, d, r) \mid \exists l', l \xrightarrow{e, g, d, r} l' \land e \in B \} \]

In [4] we considered three different TADs parallel composition rules - standard TADs, Sparse TADs and TADs with Minimal Priority Escape Transitions. The first of these coincides with Bornot and Sifakis’s definition [2, 3], while the latter two were developed by us. We argued in favour of the latter two, since standard TADs composition generated too many escape transitions. Here we re-iterate the Sparse TADs and TADs with Minimal Priority Escape Transitions definitions.

**Sparse TADs.** This is a minimal TADs approach, in which we do not generate any escape transitions. The following parallel composition (denoted \( ||^s \)) rules are used:

\[
\begin{align*}
&u[i] \xrightarrow{x, g_i, d_i, r_i} u[i]' & u[j] \xrightarrow{x, g_j, d_j, r_j} u[j]' & u[i]' & x \in CA \\
||^s u \xrightarrow{x, g', d', r', i, j} ||^s u[i'/i, j'/j] & ||^s u \xrightarrow{x, g, d, r} ||^s u[i'/i]
\end{align*}
\]

where \( 1 \leq i \neq j \leq |u|, g' = g_i \land g_j \) and \( d' = g' \land (d_i \lor d_j) \).

These rules prevent uncompleted actions from arising in the composite behaviour; they only arise in the generation of completed actions, while (already) completed actions offered by components of the parallel composition can be performed independently. This definition has the same spirit as the normal TA rules of parallel composition. The difference being that here we have deadlines which we constrain during composition to preserve the property \( d \Rightarrow g \), and hence to ensure time-reactivity.

Furthermore as a consequence of these characteristics of sparse TADs we have revised the interpretation of synchronisation in the manner we proposed. For example, if we consider again the Dying Dining Philosophers illustration, the obvious TADs formulation of **Aris1** and **Rest1**, are **Aris2** and **Rest2** shown in figure 2. Now sparse TADs composition of the two TADs yields the behaviour shown on the right of figure 2, which is action locked. This is the outcome that we were seeking. Since the pick synchronisation is not enabled, urgency cannot be enforced. This is reflected in both the guard and deadline in figure 2 being *false*.

**TADs with Minimal Priority Escape Transitions.** The idea here is the same as standard TADs, but rather than just giving escape transitions lower priority than their corresponding synchronisation, we also give them lower priority than other completed transitions. Thus, a component can only perform an escape transition if the component will never be able to perform a completed transition. This seems appropriate as our view of escape transitions is that they should only be performed...
as a very last resort - when the choice is between performing them or reaching an “error” state.

Letting, $1 \leq j \neq i \leq |u|$, the parallel composition (denoted $\parallel^m$) rules are:

$$\begin{align*}
(R1) & \quad \frac{u[i] \xrightarrow{x?,g_i,d_i,r_i} u[i']} u[j] \xrightarrow{x_1,g_i,d_j,r_j} u[j']}{||^m u \xrightarrow{x,g',d',r,r_j} ||^m u[i'/i,j'/j]} \\
(R2) & \quad \frac{u[i] \xrightarrow{x,g,d,r} u[i'] \quad x \in CA}{||^m u \xrightarrow{a,g,d,r} ||^m u[i'/i]} \\
(R3) & \quad \frac{u[i] \xrightarrow{a,g,d,r} u[i'] a \in HA}{||^m u \xrightarrow{a,g'',d'',r} ||^m u[i'/i]}
\end{align*}$$

where $g' = g_i \land g_j$, $d' = g' \land (d_i \lor d_j)$ and $1 \leq k \neq i \leq |u|$ in,

$$\begin{align*}
g'' &= g \land \{ \square - q.2 \mid q \in \theta_{CA}(u[i]) \} \land \\
    &\land \{ \square - (q.2 \land q'.2) \mid q \in \theta_{HA}(u[i]) \land q' \in \theta_{\{q,1\}}(u[k]) \} \\
d'' &= d \land g''
\end{align*}$$

(R1) is the normal synchronisation rule; (R2) defines interleaving of completed transitions; and (R3) defines interleaving of incomplete, i.e. escape, transitions. In this final rule, $g''$ holds when, (1) $g$ holds; (2) it is not the case that an already completed transition from $u[i]$ could eventually become enabled; and (3) it is not the case that an incomplete transition (including $a$ itself) offered at state $u[i]$ could eventually be completed. Furthermore, the definition ensures that $d \Rightarrow g$ and thus that time reactivity is preserved. In addition, we again obtain the “weaker” handling of urgency in synchronisation that we seek.

4. ACTION LOCKS

The last section and the TADs framework in general provide a means to compose automata together without generating timelocks. This then raises the issue of whether the same can be done for action locks.

It is clear that independent parallel composition (both in an untimed and a timed setting) preserves action lock freedom (see [5] for a formal justification). However, interaction free parallel composition is of limited value. Thus, here we consider how the same action lock compositionality property can be obtained but while allowing interaction between processes. Our definition builds upon TADs with Minimum Priority Escape Transitions.

Consider the parallel composition $\parallel^a A$ where $A$ is a vector of TADs in which the component automata have disjoint clock sets. This is necessary to avoid action locks arising as a result of component automata resetting the clocks used by other components (see [5] for further justification). Letting, $1 \leq i \neq j \leq |u|$, the product rules for $\parallel^a A$ are,
is the (now) familiar "conjunctive" synchronisation rule, with
the deadline constraint ensuring that \( d \Rightarrow g \) and thus preserving
time reactivity. (RIA) gives the also familiar interleaved modelling of inde-
pendent parallelism. (RHA) generates escape transitions in order to
avoid action locks, with the guard and deadline constructions control-
ling when the escape transitions can occur. We justify our guard and
deadline definitions now.

The guard in (RHA). This is a disjunction between the guard con-
struction for escape transitions presented in section 2 and the deadline
\((d'')\). We justify the guard based disjunct (i.e. the first) here. A later
point justifies disjoining with the deadline.

The basic idea of this first disjunct, is to enable the product to escape
action locks resulting from mismatched synchronisations. As a simple
illustration of this consider \( A_0 \) and \( A_1 \) in figure 3. Both of these TADs
are action lock free. However, if just rules (RCA) and (RIA) are used
the composition of \( A_0 \) and \( A_1 \) will action lock immediately as neither
synchronisation can be fulfilled. However, application of rule (RHA)
in conjunction with (RCA) and (RIA) will allow the action lock to be
escaped, as shown in composition (i) in figure 3.

The deadline in (RHA). The definition of \( d'' \) has a similar shape
to the guard construction we just considered, however, the temporal
operators are not included. The construction states that, the deadline
\((d'')\) of the escape transition holds if and only if,

1 the deadline of the corresponding component transition \((d)\) holds;

2 no internal transition of the component is at its deadline; and
3 no synchronisation which includes a half action of the component is at its deadline.

The intuition behind the rule is that any (non competing) deadline that appears in the component but that does not arise in the product (because of a failed synchronisation) has its deadline preserved in an escape transition of the product. A deadline of a transition is competing at a state if the deadline of an alternative transition also holds there.

This deadline construction is motivated by the observation that in the majority of cases it is the deadline that ensures action lock freeness. For example, although the automaton A in figure 3 is strongly connected it is not action lock free. In particular, assuming s0 is first entered with t==0, if it stays in state s0 for longer than 5 time units, it will action lock. Furthermore, there is nothing constraining the length of time the automaton can idle in state s0 as the deadline of the aaa! transition is false. However, (assuming s0 is entered with t<=5) if the false deadline is replaced by, say, t==5, then it would be action lock free.

Now in order to obtain the action lock freeness property that we desire we need to guarantee that deadlines that ensure action lock freeness
of component automata are preserved in the product (either through appearing as a result of rules \((RCA)\) or \((RIA)\) or by including relevant escape transitions). Our rule does this. Consider the two action lock free automata \(B\) and \(B'\) shown in figure 3. With just rules \((RCA)\) and \((RIA)\) the product of \(B\) and \(B'\) would be action locked. However, with \((RHA)\) as well, the product automaton (ii) shown in figure 3 would result.

In fact, this product would have resulted from application of the rules presented in section 2 where the deadline is simply \(d'' = d \land g''\). However, the example in figure 3 of two more action lock free TADS (\(C_0\) and \(C_1\)) shows that this is not sufficient in the general case. This is because according to the rules of section 2, the parallel composition of \(C_0\) and \(C_1\) would be as shown in figure 3 (iv) which will action lock at state \(s_1 \tau_1\).

The problem is that the guards of the \(aaa!\) and \(bbb!\) escape transitions that the rules of section 2 generate, are false. This is because in both automata an internal action can eventually be taken and this internal action will take priority.

However, if we apply the rules \((RCA)\), \((RIA)\) and \((RHA)\) of \(\parallel a\) then a product “behaviourally equivalent” to figure 3 (iii) results. This is because the deadline prevents clock \(t\) passing 5 and clock \(r\) passing 8. Notice that the guard has been pruned to match the deadline. This ensures that the enabling of \(aaa!\) and \(bbb!\) is minimised to only what is required to preserve the desired action lock freeness property.

Also notice that this example illustrates why the priority enforced in the deadline has to be immediate and including temporal operators is inappropriate. Specifically, if a deadline \(d\) ensures action lock freedom then even if later transitions are possible the deadline must be preserved exactly in the product in order to prevent later transitions from being enabled which allow an action lock to be reached, e.g. the internal transitions in \(C_0\) and \(C_1\) above. This may not be the most refined solution since we might add an escape transition even though a later transition may prevent the action lock. However, it is not currently clear how to improve upon the approach.

Finally, we need to disjoin the deadline in \(g''\) in order to ensure that \(d \Rightarrow g\) and thus to preserve time reactivity. For example, without such a disjunct, the product of \(C_0\) and \(C_1\) would timelock when \(t\) reaches 5 as the guard on the \(aaa!\) transition from \(s_0 \tau_0\) would be \(false\).

The central result of this section is given in the next theorem, it states that \(\parallel a\) preserves action lock freeness. The proof of this result is not straightforward and due to space considerations it is not possible to include it here. However, the necessary theory, the full proof and illustration of use of the operator can be found in [5], which is available on the WWW.
**Theorem 1**

\[ \exists i (1 \leq i \leq |A|). A[i] \text{ is action lock free} \Rightarrow \|a^A A \text{ is action lock free.} \]

5. **CONCLUSIONS**

This paper builds from [4] by refining its timelock results and extending the results to action lock freeness. Although related to the work of Bornot and Sifakis [2, 3], our work is different. In particular, giving escape transitions lower priority than completed actions of a particular component, is unique to our work. Furthermore, such an interpretation is important since as we have argued, real-time structures such as timeouts are inappropriately expressed with the standard TADs parallel composition operator. In addition, in obtaining our compositionality results, the only constraint we impose on component automata is that they are time and / or action lock free. In contrast, Bornot and Sifakis require a number of well behavedness criteria to hold. This limits the generality of their approach when compared with ours.

**Acknowledgements.** The author has benefited greatly from discussions with Sebastian Bornot, Joseph Sifakis and Stavros Tripakis.

**REFERENCES**


COMPILING REAL-TIME SCENARIOS INTO A TIMED AUTOMATON*

Aziz Salah, Rachida Dssouli, Guy Lapalme
Département d’Informatique et de Recherche Opérationnelle, Université de Montréal, H3T 1J4 Montreal, PQ, Canada
{salah,dssouli,lapalme}@iro.umontreal.ca

Abstract In this paper, we aim at synthesizing an executable specification for a real-time system by integrating real-time scenarios into a timed automaton. A scenario represents a partial description of a system behavior. A formal semantics is given for the model of a scenario and is used to compile a scenario into a timed automaton. The compilation algorithm is generalized to integrate several scenarios into a single timed automaton which simulates the behaviors specified by the scenarios. The results of the compilation algorithm are independent of the order in which the scenarios are added.

Keywords: real-time system, timed automata, formal specification, scenarios integration

1. INTRODUCTION

A real-time reactive system interacts with its environment under strict timing constraints. The formal specification of such systems is a difficult task that often designers prefer to avoid to move directly from informal specifications to implementation. The detection of design errors and thus the reliability of the system become difficult to evaluate increasing the development costs.

A scenario is a natural means to specify interactions of a system with its environment. The use of scenarios reduces the complexity of systems’ specification since a scenario specifies a partial behavior of the system. Our main motivation is to ease the specification of real-time reactive systems by automating some activities in the specification process. Our approach consists of synthesizing an executable specification from a set

*Partially supported by France Telecom
of provided scenarios. The resulting system specification represents a prototype in the form of a timed automaton [AD94].

Most approaches based on scenarios [HSG+94, Gli95, SDLV00] concentrate on the requirements acquisition phase at the very beginning of the system development process. Scenario based approaches may be compared based on many criteria like, for instance, the input scenarios formalism, the method of integrating scenarios and its target formalism. The method of integration is the most important criterion since any approach is based on it. The integration of scenarios consists of merging many scenarios into a global model expressed in the target formalism. The main challenge is how to characterize the steps of a scenario so that occurrences of the same steps can be identified in other scenarios. Some approaches use a manual labeling to identify scenarios’ steps while others use regular grammar[HSG+94]. Our approach uses the values of variables, which define the state of the system, to characterize scenarios’ steps. We consider two kinds of variables, discrete variables describing system properties and continuous variables (clocks) measuring time.

A scenario is defined by sequences of interactions between the system and its environment. This work is limited to sequential systems in which only one interaction is allowed to be performed at a time. We consider two basic types of interactions namely the reception and the transmission of a message which are both viewed as observable actions. We define a model of scenarios and then we define its formal semantics which is used for compilation. To compile a single scenario into a timed automaton, the scenario is first transformed into an intermediate flat format which we will refer to by the set of rule-actions of the scenario. When several scenarios are integrated, the set of all rule-actions resulting from all scenarios is compiled into a single timed automaton. Our method of scenario integration is insensitive to the order of adding scenarios so the specification of a system can be built incrementally by adding scenarios to the current prototype.

The rest of the paper is organized as follows. In section 2, we present preliminaries and notations related to the definition of the timed automata model and discrete variables of a system. In section 3, we describe the syntax of a system specification, the formal semantics of a scenario and define the timed automaton of a scenario. Section 4 is dedicated to the construction algorithm of the scenario timed automaton. We illustrate, in section 5, the use of our algorithm to synthesize a timed automaton of a scenario. Section 6 generalizes this algorithm to compile several scenarios into one timed automaton and discusses the approach.
2. PRELIMINARIES

This section is dedicated to definitions and related notations. We first recall briefly the timed automata model and semantics, and then we present system discrete variables syntax.

2.1. Timed Automata

Timed Automata (TA) model uses dense model of time for modeling timed control. TA [AD94] is an extended labeled transition system with a finite number of real variables called clocks. Clocks values increase at the same rate. Let $H$ be the set of clock variables and $\Phi(H)$ the set of clock constraints. Each $\phi \in \Phi(H)$ is defined by the grammar $\phi ::= x \# c | x - y \# c | \phi \land \phi | True$, where $x, y$ are clocks in $H$ and $\# \in \{\leq, <, =, \geq, >\}$ and $c$ a constant in $\mathbb{N}$. A clock interpretation $\theta$ is a mapping from $H$ to the set of non-negative reals. $\Theta(H)$ denotes the set of clock interpretations. For each $\theta \in \Theta(H)$ and a clock constraint $\phi \in \Phi(H)$, $\phi(\theta)$ is a boolean value describing whether $\theta$ satisfies $\phi$ or not. Given a non negative real $d$, $\theta + d$ denotes the clock interpretation which maps each clock $x$ to the value $\theta(x) + d$.

For $\lambda \subset H$, $\theta[\lambda]$ is the clock interpretation which assigns the value 0 to each $x \in \lambda$ and agrees with $\theta$ over the other clocks. $\lambda$ represents a clock assignment. Given a clock constraint $\phi \in \Phi(H)$, $\phi[\lambda]$ denotes the clock constraint satisfied by all interpretations $\theta[\lambda]$ such that $\theta$ satisfies $\phi$.

**Definition 1.** Each Timed Automaton (TA) is a tuple $A = (L_A, L^o_A, M_A, T_A, H_A)$ where,

- $L_A$ is a finite set of locations
- $L^o_A \subset L_A$ is a set of initial locations
- $M_A$ is a finite set of labels
- $H_A$ is a finite set of clocks
- $T_A \subset L_A \times M_A \times \Phi(H_A) \times 2^{H_A} \times L_A$ is the set of transitions
- $Inv$ is a mapping that assigns to each location in $L_A$ a clock constraint from $\Phi(H_A)$. This constraint is called the invariant of the location.

**Definition 2.** A labeled transition system is a tuple $\Sigma = (Q, Q_0, \rightarrow)$ where $Q$ is a set of states, $Q_0$ is the set of initial states and the relation $\rightarrow \subset Q \times Act \times Q$ is the set of transitions, where $Act$ denotes the set of labels. We write $q \xrightarrow{a} q'$ iff the transition $(q, a, q') \in \rightarrow$. 

To define the semantics of a TA $A$, a labeled transition system $\Sigma_A$ is associated with it. Each state of $\Sigma_A$ is a pair $(s, \theta) \in L_A \times \Theta(H_A)$ such that $\theta$ satisfies $Inv(s)$. $\Sigma_A$ has two types of transitions:

- transitions modeling a time elapsing of a duration $d$: $(s, \theta) \xrightarrow{d} (s, \theta + d)$, such that for all $d' \leq d$, $\theta + d'$ satisfies $Inv(s)$, and

- transitions $(s, \theta) \xrightarrow{m} (s', \theta[\lambda])$ such that $(s, m, \phi, \lambda, s') \in T_A$, $\theta$ satisfies $\phi \land Inv(s)$ and $\theta[\lambda]$ satisfies $Inv(s')$.

In a discrete transition system, traces describe system runs. For a TA $A$, the notion of next state has no meaning, in fact, a transition $e \xrightarrow{d} e'$ of $\Sigma_A$ models that the system passes through all states $e + d'$ such that $0 \leq d' \leq d$. We introduce now the notion of step as in [HNSY94, ACD93] to define a TA run.

**Definition 3.** Let $A$ be a TA, $\Sigma_A$ its associated transition system and $e, e'$ two states of $\Sigma_A$. $e'$ is reachable in one step from $e$, denoted $e \xrightarrow{d,m} e'$, if $\Sigma_A$ allows the transitions $e \xrightarrow{d} e + d$ and $e + d \xrightarrow{m} e'$ for $d \geq 0$ and $m \in M_A \cup \{0\}$. A run $\sigma$ is a (finite or infinite) sequence of steps $\sigma = e_0 \xrightarrow{d_1,m_1} e_1 \xrightarrow{d_2,m_2} e_2 \xrightarrow{d_3,m_3} \ldots$

**Definition 4.** let $\sigma = e_0 \xrightarrow{d_1,m_1} e_1 \xrightarrow{d_2,m_2} e_2 \xrightarrow{d_3,m_3} \ldots$ be a run. The timed trace of $\sigma$ is defined as the sequence $((\theta_1, l_1), \ldots, (\theta_j, l_j), \ldots)$ such that if $e \xrightarrow{d} e'$ the $j$th element of $\sigma$ for which $e = (s, \theta)$ and $m \in M_A$ then $\theta_j = \theta + d$ and $l_j = m$.

### 2.2. Discrete Variable Constraints and Assignments

Let $v$ be a system discrete variable in $V$. Let $Dom(v)$ denote a finite and discrete domain of $v$ values. Let $\Psi(V)$ denote the set of variable constraints $\psi$ defined by the grammar $\psi ::= v\#c \mid \psi \land \psi \mid \neg\psi \mid True$, where $v \in V, c \in Dom(v)$ and $\#$ is a binary relation of $Dom(v) \times Dom(v)$. Let $\Omega(V)$ be the set of variable interpretations. Each variable interpretation $\omega \in \Omega(V)$ maps a value in $dom(v)$ to each variable in $V$. $\Psi(\omega)$ denotes a boolean value describing whether $\omega$ satisfies $\psi$ or not.

Variable assignments allow variable modifications as a transformation functions. For example, given $v_1, v_2 \in V$ and an assignment $\delta = \{v_1 := v_2 + 3, v_2 := 1\}$ and a variable interpretation $\omega \in \Omega(V)$, $\omega[\delta]$ denotes a new variable interpretation defined by $\omega[\delta](v_1) = \omega(v_2) + 3$, $\omega[\delta](v_2) = 1$ and $\omega[\delta](v) = \omega(v)$ for $v \in V - \{v_1, v_2\}$. There is no limitation on assignment statements expressions but the resulting value of each variable
must remain in its domain. We write $\Delta (V)$ to denote the set of variable assignments. Given a constraint variable $\psi \in \Psi (V)$ and an assignment $\delta \in \Delta (V)$, the constraint $\psi[\delta] \in \Psi (V)$ designates the constraint satisfied by $\omega[\delta]$ such that the variable interpretation $\omega \in \Omega (V)$ satisfies $\psi$. Figure 1 shows the example of a description of a set $V$ and a set $H$ for a telephone switch controller system.

Figure 1. An example of discrete variables set $V$ and clocks set $H$

In the remaining part of this document we’ll adopt the following notations: $\theta$ denotes a clock interpretation in $\Theta (H)$, $\omega$ represents a variable interpretation in $\Omega (V)$, $\varphi$ and $\phi$ denote clock constraints in $\Phi (H)$, $\Psi$ represents a variable constraint in $\Psi (V)$, $\delta \in \Delta (V)$ denotes a variable assignment and $\lambda \subset H$ represents a clock assignment.

3. SPECIFYING A REAL-TIME REACTIVE SYSTEM BEHAVIOR

A real-time reactive system is a process characterized by a continuous interaction with its environment under strict timing constraints. This work is restricted to sequential processes. Process behaviors are specified by a set of scenarios. Each scenario is represented by sequences of actions. An action in a scenario describes the states of the process before and after the execution of the action.

Our main goal is to synthesize a TA from the provided scenarios. This TA represents a prototype of the desired system. Let $P$ be the system process. $P$ is a tuple $(S, H, V, Act)$ where $S$ is a set of scenarios, $V$ a set of discrete system variables, $H$ a set of clocks and $Act$ a set of observable labels used during interaction with the environment. The syntax for clocks in $H$ and discrete variable in $V$ have been already defined in subsections 2.1 and 2.2 respectively. In the remainder of this section, we define scenarios syntax and their semantics.

3.1. Scenario Formalization

The scenarios in $S$ describe possible behaviors of a process $P$. Each scenario action describes the changes in the process’s state after the
execution of the action or after a deadline. Each state of the process is a pair \( e = (\omega, \theta) \in \Omega(V) \times \Theta(H) \) composed of a variable interpretation \( \omega \) and a clock interpretation \( \theta \). Let us define some notations: \( \psi(e) = \psi(\omega), \varphi(e) = \varphi(\theta), e[\delta] = \omega[\delta] \) and \( e[\lambda] = \theta[\lambda] \).

**Definition 5.** A scenario \( sc \) is a tree \( sc = (N, \to, A) \) where:

- \( N = N_p \cup N_s \) such that \( N_p \) and \( N_s \) are respectively the sets of primary and secondary vertices of \( sc \). We write \( N_p = [N_1, N_2, \ldots, N_n, N_{n+1}] \), \( n + 1 = |N_p| \). Secondary vertices are leaves of the tree.
- \( A \) is the set of actions that label the tree edges. Each \( a \in A \) is a tuple \((\phi_a, \psi_a, \phi_a, \lambda_a)\) where:
  - \( \phi_a \in \Phi(H) \) and \( \psi_a \in \Psi(V) \) are constraints on the state of process.
  - \( \lambda_a \) is a clock assignment that occurs after the execution of the action \( a \).
  - \( \to \subseteq N \times A \times N \) is the set of edges.
  - Each primary vertex \( N_i \) for \( 1 \leq i \leq n : N_i = [a_{i1}, \ldots, a_{ik_i}] \) such that \((N_i, a_{i1}, N_{i+1}) \in \to \). \( a_{i1} \) is called the primary action of the vertex \( N_i \).

For \( 1 < j \leq k_i \), the actions \( a_{ij} \) of the vertex \( N_i \) are alternatives of the primary action \( a_{i1} \). These alternatives model timers expirations or any other interruption.

The semantics of a scenario action is formalized by associating with each primary vertex a pair of constraints \((\psi N_i, \varphi N_i) \in \Psi(V) \times \varphi N_i \in \Phi(H)\) defining the context of the vertex. To execute an action, the state of the process has to satisfy the context constraints of its vertex. The context constraints of a vertex are defined by the following sequence:

\[
(\psi_{N_1}, \varphi_{N_1}) = (\psi_{a_{11}}, \land_{1 < k < k_1} \varphi_{a_{1k}}), \text{ and for } 2 \leq i \leq n \\
(\psi_{N_i}, \varphi_{N_i}) = (\psi_{N_{i-1}}[\delta_{a(i-1)1}] \land \psi_{a_{i1}}, \land_{1 < k < k_i} \varphi_{a_{ik}} \land (\varphi_{N_{i-1}} \land \phi_{a(i-1)1})[\lambda_{a(i-1)1}])
\]

\(\land(\psi_{N_{n+1}}, \varphi_{N_{n+1}}) = (\psi_{N_n}[\delta_{a_n}], (\varphi_{N_n} \land \phi_{a_n})[\lambda_{a_n}])\)

Let \( a_{ij} \) be an action of \( N_i \). While the state \( e \) of the process \( P \) satisfies the condition \( \psi N_i(e) \land \psi a_{ij} \land \varphi N_i(e) \), the process \( P \) may either execute
the action $a_{ij}$ provided that $\phi_{a_{ij}}(e)$ is true, or let the time passes while the state of the process satisfies $\varphi_{N_i}$. When $P$ executes $a_{ij}$ from a state $e$, it moves to a new state $e' = (e[\delta_{a_{ij}}], e[\lambda_{a_{ij}}])$.

3.2. Timed Automaton of a Scenario

According to the expression of a context constraints of a primary vertex, the execution of an action depends on the context created by the preceding actions in the scenario. By merging scenario action and the context constraints of its primary vertex, we obtain a self contained rule-action which is independent from the execution context. A rule-action contains all the information about the states of the process and its condition of activity before and after the execution of the rule-action. Let $R(sc)$ designate the set of rule-actions resulting from the scenario $sc$. Given an action $a_{ij}$ of the vertex $N_i$, the resulting rule-action is a tuple $r = (\psi_r, \varphi_r, lab_r, \delta_r, \lambda_r, \varphi'_r)$ where:

- $\psi_r = \varphi_{N_i} \land \psi_{a_{ij}}$
- $\varphi_r = \varphi_{N_i}$
- $lab_r = lab_{a_{ij}}$
- $\delta_r = \delta_{a_{ij}}$
- $\lambda_r = \lambda_{a_{ij}}$

- For $\varphi'_r$, two cases are to be distinguished:

$$\varphi'_r = \begin{cases} \varphi_{N_{i+1}} & \text{if } 1 \leq i \leq n \text{ and } j = 1 \\ (\varphi_{N_i} \land \varphi_{a_{ij}})[\lambda_{a_{ij}}] & \text{if } 1 \leq i \leq n \text{ and } 2 \leq j \leq k_i \end{cases}$$

To define the semantics of a scenario $sc$ and its set of rule-actions $R(sc)$ we will associate to the scenario $sc$ a labeled transition system $\Sigma_{sc}$ which we refer to by the labeled transition system of the scenario $sc$, and is defined as follows:

**Definition 6.** Each rule-action $r \in R(sc)$, implies that $\Sigma_{sc}$ will contain the two following sets of transitions:

- **Immediate transitions** $e \xrightarrow{lab_r} e'$ such that $e = (\omega, \theta)$ and $e' = (\omega[\delta_r], \theta[\lambda_r])$ provided that $\psi_r(\omega) \land \varphi_r(\theta) \land \varphi_r'(\theta[\lambda_r])$ is true
- **Elapsing time transitions** $e \xrightarrow{d} e'$ such that $e = (\omega, \theta)$ and $e' = (\omega, \theta + d)$ provided that $\psi_r(\omega) \land (\forall 0 \leq d' \leq d. \varphi_r(\theta + d'))$ is true

**Definition 7.** Let $\Sigma_A$ be the labeled transition system of the TA $A$. An observational timed trace is a timed trace in $\Sigma_A$ from which all invisible labels $\varepsilon$ are removed. $OTT(e)$ denotes the set of observational timed traces that start from the state $e$ of $\Sigma_A$.

**Definition 8.** Let $\Sigma_A$ and $\Sigma_{A'}$ be the labeled transition system of the TA $A$ and $A'$ respectively. We assume that $A$ and $A'$ have the same
observational labels. \( \Sigma_A \) and \( \Sigma_{A'} \) are observational timed trace equivalent iff for each state \( e \) in \( \Sigma_A \), there exists a state \( e' \) in \( \Sigma_{A'} \) such that \( \text{OTT}(e) = \text{OTT}(e') \) and for each state \( e' \) in \( \Sigma_{A'} \), there exists a state \( e \) in \( \Sigma_A \) such that \( \text{OTT}(e) = \text{OTT}(e') \).

Theorem 1. Given a scenario \( sc \in S \), there exists a TA called \( TA \) of the scenario \( sc \), noted \( A_{sc} \), such that its labeled transition system \( \Sigma_{A_{sc}} \) is observational timed trace equivalent to \( \Sigma_{sc} \).

The proof of this theorem will be given in the next section. A location of the TA \( A_{sc} \) is characterized by a pair \((\omega, \varphi)\) where \( \varphi \) represents the invariant of the location \((\omega, \varphi)\). Each location \((\omega, \varphi)\) includes all of the states \((\omega, \theta)\) such that \( \theta \) satisfies \( \varphi \). The rule-actions of a scenario are not necessarily pairwise-disjoint (i.e. for a given scenario \( sc \), there may exist two rule-actions \( r_1, r_2 \in R(sc) \), \( \omega \) and \( \theta \) such that \( \psi_{r_1}(\omega) \wedge \psi_{r_2}(\omega) = \text{True} \) and \( \varphi_{r_1}(\theta) \wedge \varphi_{r_2}(\theta) = \text{True} \)). Let us consider the set composed of locations \((\omega, \varphi_r)\) and \((\omega[\delta_r], \varphi'_r)\) for all \( r \in R(sc) \). Each element of this set must be split into new elements so that in the resulting set \( L_{A_{sc}} \) no two elements will have a common state. The set \( L_{A_{sc}} \) is the set of locations for the TA of the scenario. Let us now define its set of transitions \( T_{A_{sc}} \) which is composed of two types of transitions:

- transitions resulting from a rule-action \( r \in R(sc) \) like \(((\omega, \varphi), \text{lab}_r, \phi_r, \lambda_r, (\omega[\delta_r], \varphi'))\) provided that:
  - \((\omega, \varphi) \in L_{A_{sc}}, (\omega[\delta_r], \varphi') \in L_{A_{sc}}\) such that \( \psi_r(\omega) \) is true, and
  - \( \varphi \wedge \varphi_r = \varphi' \) and \( \varphi' \wedge \varphi'_r = \varphi' \)
- transitions labeled by \( \varepsilon \) which we consider as the invisible label or the empty word. These transitions guarantee the time progress.

4. CONSTRUCTION OF THE TIMED AUTOMATON OF A SCENARIO

Given a scenario \( sc \), the construction of its TA \( A_{sc} \) takes two phases. First, the locations resulting from rule-actions of \( R(sc) \) are split to produce \( L_{A_{sc}} \), and then transitions are added according to the rule-actions of \( R(sc) \) to build the final \( T_{A_{sc}} \).

4.1. Phase 1: Construction of the Locations Set of \( A_{sc} \)

To split the locations resulting from \( R(sc) \), we’ll use an efficient algorithm based on the generic algorithm proposed by Bouajjani and al. [BFH + 92] to obtain a minimal split. The generic algorithm generates a minimal transition system directly from an implicit description (like a
This algorithm is useful in our context since our rule-actions are implicit description of the labeled transition system of a scenario. Let us briefly describe this algorithm and adapt it in order to construct the locations set $L_{A_{sc}}$.

Let $(Q, Q_0, \rightarrow)$ be a labeled transition system. For a partition $\rho$ of $Q$ and a state $q \in Q$, $\text{post}_\rho(q)$ denotes the set of $\rho$ elements that are immediately reachable from $q$. The function $\text{post}_\rho$ is extended to the subsets of $Q$ by $\text{post}_\rho(C) = \bigcup_{q \in C} \text{post}_\rho(q)$ where $C \subseteq Q$.

Given a class $C \in \rho$, $C$ is said to be stable with respect to the partition $\rho$ if $\forall C' \in \rho$:

$((\exists q_1 \in C, \exists q'_1 \in C' \cdot q_1 \xrightarrow{a} q'_1) \Rightarrow (\forall q_2 \in C \exists q'_2 \in C' \cdot q_2 \xrightarrow{a} q'_2))$.

The minimization algorithm uses $\text{split}(C, \rho)$ operator which splits the class $C$ into stable classes with respect to $\rho$ such that:

$\text{split}(C, \rho) = \{C_1, \ldots, C_k \mid (\forall 1 \leq i, j \leq k \cdot C_i \cap C_j = \emptyset) \text{ and } (C_1 \cup \cdots \cup C_k = C) \text{ and } (\forall i \cdot C_i \text{ stable}) \text{ and } (\forall i, j \cdot C_i \cup C_j \text{ non stable})\}$

The minimization algorithm (figure 2) progressively refines the partition $\rho$ starting from the initial partition $\rho_0$. Acc denotes the set of reachable states from the initial states. Stb contains the stable classes with respect to $\rho$. The algorithm stops when all of the reachable classes are stable with respect to $\rho$.

To apply the minimization algorithm let us consider the TA $A$ defined by:

- $H_A = H$
- $M_A = \{\text{lab}_r \cdot r \in \mathcal{R}(sc)\}$
- $L_A = \{(\omega, \varphi_r) \in \Omega(V) \times \Phi(H) \mid r \in \mathcal{R}(sc) \text{ and } \psi_r(\omega)\}$

$\cup \{(\omega[\delta_r], \varphi_r) \in \Omega(V) \times \Phi(H) \mid r \in \mathcal{R}(sc) \text{ and } \psi_r(\omega)\}$
The locations of $L_A$ are the resulting locations from the rule-actions of $R(sc)$. The TA $A$ contains all the immediate transitions of $A_{sc}$ but not all transitions modeling time elapsing. The TA $A$ is not minimal because the rule-actions of $R(sc)$ are not necessarily pairwise-disjoint. The locations of $L_A$ will be split by the minimization algorithm in order to build a minimal TA. A minimal TA means that for a given pair $(\omega, \theta)$, if there exists a location $(\omega, \varphi)$ in this TA such that $\psi(\omega) \land \varphi(\theta)$ is true, then this location is unique.

The minimization algorithm is applied to the labeled transition system of the TA $A$ and with the following initial $\rho_o$:

$$\rho_o = \{((\omega_r, \varphi_r), r \in R(sc)) \cup ((\omega_r[\delta_r], \varphi_r'), r \in R(sc))\}$$

Elements of $\rho_o$ are classes. $[\varphi]$ and $[\psi]$ denote the set of clock interpretations and variable interpretations respectively which satisfy $\varphi$ and $\psi$ respectively. For a class $C \in \rho$ denoted $(\psi_C, \varphi_C)$, $C$ is the set $[\psi_C] \times [\varphi_C]$.

For a constraint $\varphi \in \Phi(H)$, $[\varphi]$ is a convex polyhedron of $\mathbb{R}^{|H|}$. Let us define the operators ‘∩’ and ‘–’ on classes of $\rho$:

$$C \cap C' = (\psi_C \land \psi_C', \varphi_C \land \varphi_C')$$

$$C - C' = (\psi_C \land \neg \psi_C', \varphi_C \land \neg \varphi_C')$$

The intersection of convex polyhedrons is convex but their union is not necessarily convex. So $[\varphi_C \land \neg \varphi_C']$ may not be convex. For a class $C = (\psi_C, \varphi_C)$, $[\varphi_C]$ must be convex because $\varphi_C$ will be the invariant of a location of the TA under construction. Thus, let us define the operator $\text{convex}(\varphi)$ which transforms $[\varphi]$ into a partition of convex polyhedrons. The $\text{convex}$ operator is extended to classes by the expression

$$\text{convex}(\psi, \varphi) \overset{\text{def}}{=} \{(\psi, \varphi') | \varphi' \in \text{convex}(\varphi)\}.$$

The algorithm of $\text{split}(\rho, C)$ (figure 3) uses the function $\text{pre}_r(C)$ denoting the class of elements from which $C$ can be reached by transitions resulting from $r$.

The minimization algorithm assumes that $\rho_o$ is a partition but in our case, initial $\rho_o$ defined by the formula (1), is not necessarily a partition since the rule-actions of $R(sc)$ may be not disjoint. The classes of $\rho_o$ are reachable because they are resulting from $R(sc)$, and therefore there exists an iteration in the minimization procedure where $\rho$ becomes a partition and remains so until the end.

In the remainder of the paper, $\rho$ designates the resulting partition from the minimization procedure (ie all of $\rho$‘s classes are stable). The
set of locations $L_{A_{sc}}$ may now be explicitly described by the following expression:

$$L_{A_{sc}} = \{(\omega, \varphi) \in \Omega(V) \times \Phi(H) \mid \exists C \in \rho . \psi_C(\omega) \text{ and } \varphi = \varphi_C\}$$

### 4.2. Phase 2: Construction of the Transitions Set of $A_{sc}$

As mentioned earlier, the set of transitions $T_{A_{sc}}$ contains two types of transitions:

- transitions deducted from $R_{(sc)}$ like $((\omega, \varphi), \text{lab}_r, \phi_r, \lambda_r, (\omega[\delta_r], \varphi'))$ such that $r \in R_{(sc)}$, $\psi_r(\omega)$, $[\varphi] \subset [\phi_r]$ and $[\varphi'] \subset [\varphi_r']$ and

- $\varepsilon$-transitions modeling time elapsing and labeled by the invisible action $\varepsilon$. Timing constraints of $\varepsilon$-transitions will be defined further in the paper.

For example, given two locations $(\omega, h < 3)$ and $(\omega, h \geq 3)$ of $L_{A_{sc}}$ where $h$ is a clock of $H$. To model continuous time elapsing, we need to add the $\varepsilon$-transition $t = ((\omega, h < 3), \varepsilon, h = 3, \emptyset, (\omega, h \geq 3))$. Since the invariant of the location $(\omega, h < 3)$ is $(h < 3)$, the $\varepsilon$-transition $t$ will never be fired as the constraint $((h < 3) \land (h = 3))$ can’t never be satisfied. We also know that as soon as time leaves the zone $[h < 3]$, it enters to the zone $[h \geq 3]$. To avoid this topological problem, the invariant of the location $(\omega, h < 3)$ is relaxed to $(\omega, h \leq 3)$. For preserving the same transition system, the previous constraint $h < 3$ is added to the guarding constraint of the ingoing and the outgoing transitions of the relaxed location. The complete description of adding $\varepsilon$-transitions and

```plaintext
function split(\rho, C)
    Z := \{C\}
    for each $C' \in \rho$
        for each $r \in R$
            $P := \emptyset$
            $U := pre_r(C')$
            for each $X \in Z$
                $W := U \cap X$
                if $(W = \emptyset \text{ or } W = X)$ then $P = P \cup \{X\}$
                else $P := P \cup \{W\} \cup \text{convex}(X - W)$
            $Z := P$
        return $(Z)$
```

Figure 3. Split function
relaxing locations is given by functions add_ε-transition and Relax_Inv (figure 4). These algorithms use the following definitions:

**Definition 9.** Given $\varphi \in \Phi(H)$, the relaxed constraint of $\varphi$ denoted $\overline{\varphi}$, is the clock constraint in which the relation $<$ and $>$ are respectively replaced by $\leq$ and $\geq$ in the expression of $\varphi$.

**Definition 10.** We write $\varphi_1 \prec \varphi_2$ iff $[\varphi_1] \cap [\varphi_2] = \emptyset$ and $([\varphi_1] \cap [\overline{\varphi_2}] \neq \emptyset$ or $[\overline{\varphi_1}] \cap [\varphi_2] \neq \emptyset$)

The resulting sets $T_{A_{sc}}$ and $L_{A_{sc}}$ after the add_ε-transition function call represent respectively the set of transitions and the set of locations of the TA $A_{sc}$. The set of labels is $M_{A_{sc}} = M_A \cup \{\varepsilon\}$.

### 4.3. Observational Timed Trace Equivalence

$\Sigma_{sc}$ and $\Sigma_{A_{sc}}$ are the labeled transition systems of the scenario $sc$ and its TA $A_{sc}$ respectively. $A_{sc}$ may contain some $\varepsilon$-transitions while $\Sigma_{sc}$ does not. $\Sigma_{A_{sc}}$ and $\Sigma_{sc}$ must be equivalent according to theorem 1. The objective of this equivalence is to allow us to ignore $\varepsilon$ labels in the timed traces of $A_{sc}$ to be able to compare them with the timed traces of $\Sigma_{sc}$ since $\varepsilon$-transitions are not observable by the environment. Let us first define what we mean by runs and timed traces for $\Sigma_{sc}$. If we consider that $\Sigma_{sc}$ is a labeled transition system of a fictive TA, the runs and timed traces of $\Sigma_{sc}$ may be defined exactly as we did previously in definitions.
3 and 4 for a TA. The next proposition defines an equivalence between $\Sigma_{sc}$ and $\Sigma_{A_{sc}}$:

**Proposition 1.** $\Sigma_{A_{sc}}$ and $\Sigma_{sc}$ are observational timed trace equivalent.

**Proof.** (sketch) Given a state $e$ of $\Sigma_{sc}$ there exists $r \in R(sc), \omega$ and $\theta$ such that $e = (\omega, \theta)$ and at least one of the two following cases is true: 1) $\psi_r(\omega)$ and $\varphi_r(\theta)$, or 2) $\omega = \omega' [\delta_r]$ and $\psi_r(\omega')$ and $\varphi_r'(\theta)$. The states of $\Sigma_{A_{sc}}$ are in the form of $((\omega, \varphi), \theta)$ since $(\omega, \varphi)$ may represent a location in $A_{sc}$. There exists a state $e' = ((\omega', \varphi'), \theta)$ of $\Sigma_{A_{sc}}$ resulting from the split of either the class $(\psi_r, \varphi_r)$ or $(\psi_r[\delta_r], \varphi_r')$. One can then proof $OTT(e) = OTT(e')$. □

According to the previous proposition, $\Sigma_{sc}$ et $\Sigma_{A_{sc}}$ are equivalent from the point of view of observational timed trace equivalence. The construction of $A_{sc}$ and the proposition 1 provide a proof of theorem 1.

The resulting partition $\rho$ from the construction of $A_{sc}$ allows to reduce the combinatorial explosion of the locations in $A_{sc}$. $\rho$ abstracts $A_{sc}$ into a compact TA $A_{sc}/\rho$ such that:

- $L_{A_{sc}/\rho} = \rho$
- $L^0_{A_{sc}/\rho} = \{ C \in \rho | C \cap L^0_A \neq \emptyset \}$
- $T_{A_{sc}/\rho} = \{ (\langle C, m, \phi, \lambda, C' \rangle) \mid (s, m, \phi, \lambda, s') \in T_{A_{sc}}, s \in C \text{ and } s' \in C' \}$

A location of the TA $A_{sc}/\rho$ groups together all the location in $A_{sc}$ which are imperceptible if we do not take into account the values of the discrete variable. Such representation is more suitable as an input to verification tools since it is more compact than the initial TA $A_{sc}$.

5. **APPLICATION: TELEPHONE SWITCH CONTROLLER**

This section illustrates the construction of a TA from a simple scenario. The process $P$ models a telephone switch controller. Discrete variables set $V$ and clocks set $H$ are those described in figure 1. In this example, we will construct the TA of the following call establishment scenario:

“When the user $A$ is idle, if the controller receives the pickup($A$) message, it sends the tone to user $A$. If user $A$ calls user $B$ before 30 time units then the controller sends the ring to $B$ provided that he’s idle. But if user $A$ does nothing during 30 time units, the controller sends him the busy_tone($A$) message. While the user $B$ terminal is ringing, if user $B$ picks up before 60 time units, the controller establishes the call, else the later is canceled and the controller sends busy_tone($A$)”.
The representation of the tree of this scenario was removed from this document, for lack of space, but it can be restored from the first column of table 1 according to definition 5. Table 1 describes the set of rule-actions resulting from the call establishment scenario. The construction of the TA scenario uses as input the rule-actions of table 1 and its initial distribution classes $\rho_0$. The abstraction of the scenario TA defined by the resulting partition $\rho$ is drawn in figure 5.

<table>
<thead>
<tr>
<th>$a_{ij}$</th>
<th>Lab</th>
<th>$\varphi_r$</th>
<th>$\psi_r$</th>
<th>$\lambda_r$</th>
<th>$\varphi'_r$</th>
<th>$\rho_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
<td>True</td>
<td>pickup(A)</td>
<td>$A_{stat}=IDLE \land A_{sig}=NONE$</td>
<td>True</td>
<td>$A_{stat}=BUSY$</td>
<td>${h}$</td>
</tr>
<tr>
<td>a21</td>
<td>$h=0$</td>
<td>send_tone(A)</td>
<td>$A_{stat}=BUSY \land A_{sig}=NONE$</td>
<td>$h=0$</td>
<td>$A_{sig}=TONE$</td>
<td>${h}$</td>
</tr>
<tr>
<td>a31</td>
<td>$h&lt;30$</td>
<td>dialing(B)</td>
<td>$A_{stat}=BUSY \land A_{sig}=TONE$</td>
<td>$h&lt;30$</td>
<td>$A_{sig}=DIALING(B)$</td>
<td>${h}$</td>
</tr>
<tr>
<td>a32</td>
<td>$h&lt;30$</td>
<td>busy_tone(A)</td>
<td>$A_{stat}=BUSY \land A_{sig}=TONE$</td>
<td>$h=30$</td>
<td>$A_{sig}=BUSY_TONE$</td>
<td>${h}$</td>
</tr>
<tr>
<td>a41</td>
<td>$h=0$</td>
<td>ring(A,B)</td>
<td>$A_{sig}=DIALING(B) \land A_{stat}=BUSY \land B_{stat}=IDLE \land B_{sig}=NONE$</td>
<td>$h=0$</td>
<td>$A_{sig}=ECHO_RING(B), B_{sig}=RING(A)$</td>
<td>${h}$</td>
</tr>
<tr>
<td>a51</td>
<td>$h\leq 60$</td>
<td>pickup(B)</td>
<td>$A_{sig}=ECHO_RING(B) \land D_{sig}=RING(A) \land A_{stat}=BUSY \land B_{stat}=BUSY$</td>
<td>$h&lt;60$</td>
<td>$A_{sig}=TALKING, B_{sig}=TALKING$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>a52</td>
<td>$h\leq 60$</td>
<td>busy_tone(A)</td>
<td>$A_{sig}=ECHO_RING(B) \land A_{stat}=BUSY \land B_{stat}=BUSY$</td>
<td>$h=60$</td>
<td>$A_{sig}=BUSY_TONE, B_{sig}=NONE, B_{stat}=IDLE$</td>
<td>${h}$</td>
</tr>
</tbody>
</table>

Table 1. Description of the rule-action set $R(sc)$ for the call establishment scenario. Each row in the table, contains the original scenario action, its corresponding rule-action and its two resulting classes in the initial $\rho_0$.

6. METHOD OF SCENARIOS INTEGRATION

The objective of the integration of scenarios is to merge many of them into a single TA. The construction of the scenario TA algorithm may be applied on any rule-actions set. Let $A_R$ be the TA synthesized from the set of rule-actions $R$. Based on this notation, the TA of a scenario $sc$ may also be denoted $A_{R(sc)}$. Particularly, for the scenarios $sc_1$ and $sc_2$, their respective TA $A_{sc_1}$ and $A_{sc_2}$ are merged into one TA by the operator $\oplus$ defined as follows:

$$A_{sc_1} \oplus A_{sc_2} = A_{R(sc_1)} \oplus A_{R(sc_2)} \overset{def}{=} A_{R(sc_1) \cup R(sc_2)}$$
Figure 5. Abstract Timed automata of the call establishment scenario. The classes, for example, $C_{11}$ and $C_{12}$ classes result from the split of the class $C_1$ and so on.

**Proposition 2.** For each state $e$ in $\Sigma_{sc}$ ($sc \in \{sc_1, sc_2\}$) there exists a state $e'$ in $\Sigma_{sc_1 \oplus sc_2}$ such that $OTT(e) \subset OTT(e')$. The reverse is false because of the possible overlapping of the scenarios.

The proposition 2 allows to construct one TA $A$ which models the process $P = (S, H, V, Act)$ and to integrate all of its scenarios. The TA $A$ results from the compilation of the rule-actions set $\bigcup_{sc \in S} R(sc)$. Given $S = \{sc_1, \ldots, sc_n\}$, the following formula describes the above stated compilation:

$$A = A_{sc_1} \oplus A_{sc_2} \oplus \cdots \oplus A_{sc_n} = A_{R(sc_1) \cup R(sc_2) \cup \cdots \cup R(sc_n)}$$

(2)

According to proposition 2, $A$ includes all the behaviors allowed by each integrated scenario. $A$ also includes some extra behaviors which may result from some overlapping scenarios. It is possible to identify these extra behaviors for an eventual validation process. The salient features of the integration algorithm may be inferred from the formula (2) as follows:

- The specification of a system is incremental. Assuming that the current specification of an existing system results from the compilation of a set of scenarios $S$, the system extension to support new services consists of adding new scenarios to the current specification. Assume $S'$ is the set of those new scenarios. The whole system prototype is now the TA $A_R \oplus A_{R'}$ where $R = \bigcup_{sc \in S} R(sc)$ and $R' = \bigcup_{sc \in S'} R(sc)$. Each intermediate prototype of the system may be checked for the detection of possible features interaction.

- The operator $\oplus$ is commutative and associative as the union $\cup$. So, the order in which scenarios are added to construct the specification doesn’t matter.
Previous compilation results are reused when a new scenario $sc$ is added. In the formula $A \bigcup_{sc \in \Delta} R(sc) \oplus A_{sc'}$, the first operand is not recomputed but just reused to get the new prototype of the system.

7. CONCLUSION

In this paper, our main contribution is the development of an algorithm for compiling many real-time scenarios into a single timed automaton which represents an executable prototype of the system. This algorithm is insensitive to the order in which scenarios are integrated.

First we have proposed a syntax and a formal semantics for scenarios. Then we defined a flat format of a scenario as a set of self contained rule-actions. The compilation algorithm synthesizes a timed automaton from a set of rule-actions. The set of rule-actions may come from more than one scenarios and therefore the resulting timed automaton represents the integration of these scenarios. The compact representation format of the timed automaton produced by our integration algorithm is compatible to use with model checking tools like KRONOS and UPPAAL.

As future work, we wish to further define operators for explicit integration of the scenarios. These operators are used as directives of compilation.

REFERENCES


DERIVING PARAMETER CONDITIONS FOR PERIODIC TIMED AUTOMATA SATISFYING REAL-TIME TEMPORAL LOGIC FORMULAS*

Akio Nakata and Teruo Higashino
Dept. of Informatics and Mathematical Science, Osaka University
Toyonaka, Osaka 560-8531, Japan
{nakata,higashino}@ics.es.osaka-u.ac.jp

Abstract A symbolic model checking method for parametric periodic timed automata is proposed. The method derives symbolically the weakest condition for parameters such that the specified control state of a periodic timed automaton satisfies some temporal properties. Unlike several existing parametric symbolic model checking methods, the proposed method is ‘on-the-fly’ — it does not unnecessarily check all the states. Instead, it traverses some necessary part of the computation tree to derive the weakest condition. We show that if we constrain a timed automaton to be periodic, i.e. if we force a timed automaton to return to its initial state periodically at the specified constant time, we have only to traverse at most the first 3 periods in the infinite computation tree. In the proposed method, we can avoid a costly (and generally undecidable) fixpoint-calculation for dense-time-domain state sets, and derive the weakest condition for parameters of a timed automaton to satisfy given temporal properties written in a real-time temporal logic formula.

Keywords: symbolic model checking, real-time periodic system, temporal logic

1. INTRODUCTION

Model checking[1] have been recognized as one of very useful and effective methods for designing reliable hardware/software systems. Especially, in recent years, real-time systems have been developed for the areas that high reliability is required, such as aircraft/train/car controlling, nuclear reactors, medical devices and other real-time systems which may be produced a lot and

---

*This work is partially supported by Semiconductor Technology Academic Research Center (STARC), Japan.
hard to modify in later (e.g. hardware chips or embedded systems). Model checking techniques may be very useful for developing such reliable real-time systems to ensure that the system’s design written in some formal model satisfies the required properties such as safety, liveness, and fairness.

The classical model checking method is not parametric, that is, to check whether a behavioral specification written in some state model, satisfies some requirement specification (property) written in temporal logic, all the parameters in the specification must be fixed to some concrete values. In recent years, several symbolic model checking methods are proposed [2, 3, 4, 5]. Symbolic model checking is a method to construct a set of states (state-sets) which satisfy given temporal properties by representing infinite or finitely huge state-sets symbolically and using a symbolic calculation. The crucial part of symbolic calculation is a fixpoint calculation. There are some symbolic representations of state-sets, such as BDDs[2], which enable us to compute fixpoints efficiently. Although BDDs themselves are aimed for compression of finitely-huge state space, if we adopt some symbolic representations such as Presburger Arithmetic[6], we can extend the method to possibly infinite state space. Such an approach is especially useful for parametric analyses, that is, we can obtain the representation of the set parameter variables in order that the behavioral specification satisfies the requirement specification, instead of repeatedly guessing concrete parameter values and checking satisfiability.

More recently, [3] proposed a semi-decision procedure to derive a symbolic representation of parametric states of a hybrid automaton (an extension of a timed automaton) in order to reach some given state-sets. They adopt first-order theory with addition on real-numbers[6] as symbolic representation of state-sets. Although the satisfiability of the first-order theory with addition on real-numbers is decidable, fixpoint calculation is very costly and generally undecidable. [4] proposed some approximation techniques to cope with such undecidability of fixpoint calculations using Presburger Arithmetic as symbolic representation of state-sets, but the domain of the variables is restricted to integers. On the other hand, [5] proposed an algorithm to obtain the condition of parameters in order that the given non-parametric state model on dense time domain satisfies the given parametric temporal logic formula. However, they only allow to write parameters to temporal logic formulas, not in a timed automaton. In a realistic system design process, we usually want to choose parameter values of models (implementations) rather than in temporal logic (specifications).

Thus, we propose a decision algorithm to derive a set of parameters of a subclass of a timed automaton model which may contain parameters (parametric timed automata[7]) and satisfies a formula of a real-time extension of CTL[1]. In our method, parameters are allowed in both a model and a temporal logic formula. We adopt formulas of first-order theory with addition on
real-numbers as symbolic representation of sets of parameter values, as similar to [3]. In order to reduce the size of the intermediate symbolic representation, we decompose the given problem on-the-fly to several subproblems, and recursively solve the subproblems to construct the entire condition for parameters. Unlike [3], in this approach, we need not encode symbolically the entire state space (it tends to be very long) and only the necessary part of the tree is traversed. Specifically, we compute the weakest condition $WPC(s, f)$ of parameters in order that state $s$ of the given model should satisfy the given temporal property $f$. First, we define $WPC(s, f)$ as a recursive function such as

$$WPC(s, f) \overset{\text{def}}{=} F(WPC(s_1, f_1), \ldots, WPC(s_k, f_k)),$$

where $F()$ is a functional on first-order formulas, each $s_i$ is either $s$ or some next state of $s$, and each $f_i$ is either a proper subformula of $f$ or some derived formula of $f$ (not necessarily a subformula of $f$). Basically, we can compute the weakest condition $WPC(s, f)$ if the application of the recursive definition of $WPC(s, f)$ is ensured to terminate. However, that is not the case in general. If the model contains some loops, such a recursive application does not terminate. To cope with the problem, we find some subclass of the timed automata such that we need not to explore the infinite computation tree. When the model is a periodic timed automaton, that is, after some fixed time period it returns to its initial state and repeats its behavior periodically, we have only to check some finite part of the infinite computation tree and can output the result.

In our method, $WPC(s, f)$ is ensured to be obtained after a fixed steps of recursive computations, thus we can avoid costly fixpoint calculations of first order theory on real-numbers. Moreover, in our on-the-fly approach, intermediate results need to compute $WPC(s, f)$ are kept small compared to the state-space construction approaches. Using dynamic programming, we can also avoid duplicate computation of $WPC(s_i, f_i)$'s when the pair of $s_i$ and $f_i$ is the same. Moreover, each subcondition $WPC(s_i, f_i)$ can be computed in parallel, so we can easily parallelize our method using parallel processors to improve the efficiency.

The rest of this paper is organized as follows. In Section 2, we introduce our model, periodic timed automata. In Section 3, we give a definition of the logic, real-time CTL. In Section 4, we explain our method to obtain $WPC(s, f)$ in detail. Section 6 concludes this paper.

**Related Works** Several parametric model checking method have been proposed for real-time models[7, 8, 9] and value-passing I/O models[10]. For given two parametric state models, Refs.[7, 8, 10] have proposed the methods to derive the parameter conditions to make one model a correct implementation of another model. [7] has adopted language inclusion as an implementation relation, while [8, 10] has adopted bisimulation equivalence. [8] is an extension
of [10] to a timed model. The proposed method in [9] takes a non-parametric state model on discrete time domain and a temporal logic formula which may have some parameters bounded by quantifiers, and checks whether the model satisfies the logic formula. Although some very interesting properties may be expressed using parameters, it simply checks whether or not the given model satisfies the given property. In addition, they consider that the time domain is integers.

2. PARAMETRIC AND PERIODIC TIMED AUTOMATA

In this section, we formally define our model, periodic timed automata. In addition to the traditional theory of Timed Automata[11], we introduce parameters on any (discrete or dense) domain. Then, we define a (non-periodic) parametric timed automaton model. Our definition of timed automata is essentially the same but slightly different from the parametric timed automaton in [7], since we generally allow timing constraints to be first-order formulas with addition on real-numbers.

Let Act, Var, Pred(Var) denote the set of all actions, the set of all variables, and the set of all formulas of first order theory with addition on real-numbers over Var, respectively. We also denote the set of real-numbers by \( \mathbb{R} \) and the set of non-negative real-numbers by \( \mathbb{R}^+ \).

**Definition 2.1** A parametric timed automaton is a tuple \( \langle S, C, PVar, E, Inv() \rangle \), where \( S \) is finite set of control states, \( C \subseteq Var \) is a finite set of clocks, \( PVar \subseteq Var \) is a finite set of parameters, \( E \subseteq S \times Act \times Pred(Var) \times 2^{C} \times S \) is a transition relation, \( Inv() : S \mapsto Pred(Var) \) is an invariant condition for each state, \( s_{init} \) is the initial state. We write \( s_i \xrightarrow{a,P,r} s_j \) if \( (s_i, a, P, r, s_j) \in E \).

Informally, a transition \( s_i \xrightarrow{a,P,r} s_j \) means that action \( a \) can be executed from \( s_i \) when the values of both clocks and parameters satisfy the formula \( P \) (called a guard condition), and after executed, the state moves into \( s_j \) and clocks in the set \( r \) are reset to zero. In any state \( s \), values of all clocks increase continuously at the same speed, representing the time passage. Note that the values of clocks (and parameters if any) can never violate the invariant condition \( Inv(s) \). Intuitively, \( Inv(s) \) represents the range of values (e.g. minimum and maximum values) allowed for clocks (and parameters). Thus, time passage at state \( s \) will stop when the value of some clock will exceed the maximum value specified by \( Inv(s) \). When time passage stops, some executable action is forced to execute, representing urgency[12] of the action. Also, any incoming transition of the state \( s' \) violating \( Inv(s') \) is not allowed.

**Example 2.1** Fig. 1 is a simple example of a parametric timed automaton. In Fig. 1, a set of parameters is \( \{x, y, z\} \), a set of clocks is \( \{c, c'\} \), the initial state
is s, and the transition \( s[c \leq x] \xrightarrow{a, [c \leq x - 2], 1} s_1[\text{true}] \) means that at state \( s \), for a given value of the parameter \( x \), some time may be elapsed (i.e. the clock \( c \) increases) while \( c \) satisfies the invariant \([c \leq x]\), and when \( c \leq x - 2 \) holds, action \( a \) can be executed, no clocks are reset to zero, and the state changes to \( s_1 \) (the invariant of \( s_1 \) is \([\text{true}]\), which means that any clock and parameter values are allowed at \( s_1 \)). Similarly, the transition \( s[c \leq x] \xrightarrow{b, [c \leq x - 3], (c')} s_2[c \leq x] \) means that some time may be elapsed while \( c \) satisfies the invariant \([c \leq x]\), and when \( c > x - 3 \) holds, action \( b \) can be executed, clock \( c' \) is reset to zero, and the state changes to \( s_2 \). In the transition \( s_2[c \leq x] \xrightarrow{d, [c \geq y \land c' \leq z], 1} s_3[\text{true}] \), a guard condition for both clocks \( c \) and \( c' \) are specified using parameters \( y \) and \( z \).

Formal semantics of timed automata is defined as follows. The values of clocks and parameters are given by a function \( \sigma : (C \cup P\text{Var}) \rightarrow \mathbb{R} \). We refer to such a function as a value-assignment. We represent a set of all value-assignments by \( \text{Val} \). We write \( \sigma \models P \) if a formula \( P \in \text{Pred}(\text{Var}) \) is true under a value-assignment \( \sigma \in \text{Val} \). The semantic behavior of a parametric timed automaton is given as a semantic transition system on concrete states. A concrete state is represented by \((s, \sigma)\), where \( s \) is a control state and \( \sigma \) is a value-assignment. Let \( \text{CS} \triangleq \{(s, \sigma) | s \in S, \sigma \in \text{Val}\} \) be a set of concrete states. The semantic transition system consists of delay-transitions and action-transitions. A delay transition represents a time passage within the same control state \( s \in S \), whereas an action transition represents an execution of an action which changes the control state to the next one \( s' \). Formally, the semantic transition system is defined as follows.

**Definition 2.2** For any value-assignment \( \sigma \), \( t \in \mathbb{R}^+ \), and \( r \subseteq C \), let \( \sigma + t \) and \( \sigma[r \rightarrow 0] \) be the value-assignments such that

\[
(\sigma + t)(x) \triangleq \begin{cases} 
\sigma(x) + t & \text{if } x \in C, \\
\sigma(x) & \text{otherwise.} 
\end{cases}
\]

\[
(\sigma[r \rightarrow 0])(x) \triangleq \begin{cases} 
0 & \text{if } x \in r, \\
\sigma(x) & \text{otherwise.} 
\end{cases}
\]
A semantic transition system for a parametric timed automaton $\langle S, C, PVar, E, Inv() \rangle_{s_{init}}$ is a labelled transition system on concrete states $CS$, where the transition relation is defined by the following rules:

- $(s, \sigma) \xrightarrow{t} (s, \sigma + t)$ if $t \in \mathbb{R}^+$ and $(\sigma + t) \models Inv(s),$
- $(s, \sigma) \xrightarrow{a} (s', \sigma[r \rightarrow 0])$ if $s \xrightarrow{a_{P,r}} s'$, $\sigma \models P$ and $\sigma[r \rightarrow 0] \models Inv(s').$

The major difference of periodic timed automata from normal parametric timed automata is that it checks the elapsed time since it is started, and if it is equal to the specified period $T$, then it resets to its initial state. Moreover, it is assumed that only finitely bounded actions can be performed before returning to the initial state. To ensure the above properties, we define a periodic timed automaton as one obtained by adding reset transitions to a parametric timed automaton with no loops (we refer to such a parametric timed automaton as a parametric timed DAG-automaton). Formally it is defined as follows.

**Definition 2.3** A parametric timed DAG-automaton is a parametric timed automaton whose transition graph has no directed cycles, (i.e. it is a Directed Acyclic Graph(DAG)).

The parametric timed automaton in Example 2.1 is a parametric timed DAG-automaton since its transition graph is a tree (so it is also a DAG).

**Definition 2.4** A periodic timed automaton is a parametric timed automaton which is obtained from a parametric timed DAG-automaton by adding the special reset transition (called a return transition) $s \xrightarrow{i_{[c_p=T]},C} s_{init}$ for state $s$, where $s_{init}$ is the initial state, $C$ is a set of all clocks, $c_p \in C$ is a special clock which keeps the elapsed time from the initial state $s_{init}$ (no other transition can reset this clock), $T \in \mathbb{R}^+$ is a period, $i \in Act$ is a special reset action.

**Example 2.2** Fig. 2 is an example of a periodic timed automaton. This example is a modified version of Example 2.1 where a special clock $c_p$ and some return transitions such as $s \xrightarrow{i_{[c_p=T]},[c,e',c_p]} s$ are added. Note that we allow periodic timed automata to terminate instead of returning to the initial state, such as the state $s_3$ in Fig. 2.

### 3. REAL-TIME CTL

In this section, we define RPCTL, a Real-time and Parametric extension of Computation Tree Logic(CTL)[1] including some operators in ACTL[13].

---

1. Unlike [1] and many timed extensions of CTL such as TCTL[3], the next operator of RPCTL is attributed by an action name (similar to Hennessy-Milner Logic[14], or ACTL[13]) so that we can verify the properties
In compared to TCTL[3], we do not adopt the freeze quantifier as a primitive operator of RPCTL.

**Definition 3.1** The syntax of RPCTL formula is defined by the BNF in Fig. 3, where \( a \in Act \) is an action name, \( p \) is a linear expression which may contain parameter variables, and \( \sim \in \{<, \leq, >, \geq, =\} \) is a comparison operator. We may omit ‘\(~ p\)’ specifier, and in that case ‘\( \geq 0\)’ is assumed.

RPCTL is a logic to specify a temporal property at the state of of a parametric timed automaton for its succeeding behavior using temporal operator with timing constraints which may contain parameters. Intuitive meaning of basic constructs of RPCTL is as follows. ‘true’ holds at any concrete state. ‘\( \neg f \)’ holds at a concrete state \((s, \sigma)\) if and only if \( f \) does not hold at \((s, \sigma)\). ‘false’ never holds at any concrete state, which is equivalent to ‘true’. ‘\( f_1 \land f_2 \)’ holds if and only if both \( f_1 \) and \( f_2 \) hold. ‘\( f_1 \lor f_2 \)’ and ‘\( f_1 \rightarrow f_2 \)’ are also defined similarly to classic propositional logic. ‘\( \langle a \rangle_{\leq p} f \)’ holds at \((s, \sigma)\) if and only if there exists some transition from \((s, \sigma)\) performing \( a \) within \( p \) units of time, such that \( f \) holds at the next (control) state. Since we can define similarly if

---

when we view the model as a mealy-machine, or (timed-extension of) process algebra with observation semantics (i.e. processes are identified by observing actions, not states) such as CCS, CSP or LOTOS.
there exists some transition sequence \( (s, \sigma) \xrightarrow{t} (s, \sigma + t) \xrightarrow{a} (s', \sigma') \)
such that \( t \sim p \) and \( (s', \sigma') \models f \).

**(s, \sigma) \models f_1 \land f_2**

\( (s, \sigma) \models f_1 \land f_2 \) is equivalent to \((s, \sigma) \models f_1 \) and \((s, \sigma) \models f_2 \).

**(s, \sigma) \models \langle a \rangle_{-p} f**

\( (s, \sigma) \models \langle a \rangle_{-p} f \) holds if there exists some transition sequence \( (s, \sigma) \xrightarrow{t} (s, \sigma + t) \xrightarrow{a} (s', \sigma') \)
such that \( t \sim p \) and \( (s', \sigma') \models f \).

**(s, \sigma) \models f_1 \text{EU}_{-p} f_2**

\( (s, \sigma) \models f_1 \text{EU}_{-p} f_2 \) holds if there exists some transition sequence

\[
(s, \sigma) = (s_1, \sigma_1) \xrightarrow{t_1} (s_1, \sigma_1 + t_1) \xrightarrow{a_1} \cdots \xrightarrow{t_{k-1}} (s_{k-1}, \sigma_{k-1} + t_{k-1}) \xrightarrow{a_{k-1}} (s_k, \sigma_k)
\]

and some non-negative real-number \( t_k \), such that \( (s_k, \sigma_k + t_k) \models f_2 \) and \( t_1 + \cdots + t_k \sim p \) and for any \( i \) (\( 1 \leq i \leq k \)) and for any \( t'_i (0 \leq t'_i < t_i) \), \( (s_i, \sigma_i + t'_i) \models f_1 \).

**(s, \sigma) \models f_1 \text{AU}_{-p} f_2**

\( (s, \sigma) \models f_1 \text{AU}_{-p} f_2 \) holds if for any transition sequence such that

\[
(s, \sigma) = (s_1, \sigma_1) \xrightarrow{t_1} (s_1, \sigma_1 + t_1) \xrightarrow{a_1} \cdots \xrightarrow{t_{k-1}} (s_{k-1}, \sigma_{k-1} + t_{k-1}) \xrightarrow{a_{k-1}} (s_k, \sigma_k)
\]

and for any nonnegative real-number \( t_k \), \( (s_k, \sigma_k + t_k) \models f_2 \) and \( t_1 + \cdots + t_k \sim p \) and for any \( i \) (\( 1 \leq i \leq k \)) and for any \( t'_i (0 \leq t'_i < t_i) \), \( (s_i, \sigma_i + t'_i) \models f_1 \).

**Figure 4. Semantics of RPCTL**

\(~\) is other than \( \leq \) (case of \( \geq, <, >, = \)), we only mention the case of \( \leq \) in the following explanation. \( [a]_{\leq p} f \) holds if and only if for any transition from the state performing \( a \) within \( p \) units of time, \( f \) holds at the next state, which is the same as \( \neg [a]_{\geq p} \neg f \). \( f_1 \text{EU}_{\leq p} f_2 \) holds if and only if there exists some transition sequence such that \( f_2 \) eventually holds within \( c \) units of time and until then, \( f_1 \) always holds. \( f_1 \text{AU}_{\leq p} f_2 \) holds if and only if for any transition sequence, \( f_2 \) eventually holds within \( p \) units of time and until then, \( f_1 \) always holds. We can also use \( \text{EF}_{\leq p} f \), \( \text{AG}_{\leq p} f \), \( \text{AF}_{\leq p} f \) and \( \text{EG}_{\leq p} f \) as abbreviations for \( \text{true \ EU}_{\leq p} f \), \( \neg \text{EF}_{\leq p} f \), \( \text{true \ AU}_{\leq p} f \) and \( \neg \text{AF}_{\leq p} f \), respectively.

In general, we write \( M, (s, \sigma) \models f \) to mean that an RPCTL formula \( f \) is satisfied by a concrete state \( (s, \sigma) \) of a parametric timed automaton \( M \). If there are no confusions, we omit \( M \) and just write \( (s, \sigma) \models f \). The formal definition of the relation \( \models \) is as follows. We only give the definitions for six primitive constructs, \( \text{true}, \neg f, f_1 \land f_2, \langle a \rangle_{-p} f, f_1 \text{EU}_{-p} f_2 \) and \( f_1 \text{AU}_{-p} f_2 \). The rest of the constructs can be specified similarly to the above constructs.

**Definition 3.2** The relation \( (s, \sigma) \models f \) is formally defined in Fig. 4.

**Example 3.1** The RPCTL formula

\[
[a_1]_{q_1} (((a_2)_{true}) \text{EU}_{q_2} ((a_3)_{true}))
\]
means that for every state reachable after executed action $a_1$ within $q_1$ units of time, there exists an execution path such that action $a_2$ is always executable until $a_3$ becomes executable after $q_2$ units of time elapsed. Note that $q_1$ and $q_2$ are parameter variables.

\[ \Box \]

4. DERIVATION OF THE WEAKEST CONDITION OF PARAMETERS

Now we describe our method to derive symbolically the weakest condition of parameters $WPC(s, f)$ in order that state $s$ of the periodic timed automaton satisfies the RPCTL property $f$. To begin with, we give a precise description of our problem.

**Definition 4.1** Let $M$ be a parametric timed automaton, $s$ be a state of $M$, and $f$ be an RPCTL formula. The parameter condition derivation problem is to derive a first-order formula $WPC(s, f)$ such that

\[ \sigma \models WPC(s, f) \iff (s, \sigma) \models f. \]

At first, we give an algorithm to solve the parameter condition derivation problem for parametric timed DAG-automata, and then we extend it to periodic timed automata.

4.1. Case of DAG Models

As mentioned in Section 1, we define $WPC(s, f)$ as a recursive function such that

\[ WPC(s, f) \overset{\text{def}}{=} F(WPC(s_1, f_1), \ldots, WPC(s_k, f_k)). \]

Here we give a concrete definition of function $WPC(s, f)$ for each construct of RPCTL formula $f$.

**Definition 4.2** Let $s$ and $f$ be a state of a parametric timed automaton and an RPCTL formula $f$, respectively. Then, function $WPC(s, f)$ is defined in Fig. 5. In Fig. 5, $C$ is a set of all clocks, $r, r_i, \text{etc.}$ denote subsets of clocks, $P[C + t/C]$ ($P[0/r]$) represents a first order formula $P$ whose every free occurrence of each variable $x \in C \ (x \in r)$ is replaced with $x + t$ ($0$, respectively).

The meaning of the definition of $WPC(s, f)$ is as follows. If $f$ is one of $\text{true}$ or $f_1 \land f_2$, the definition of $WPC(s, f)$ is straightforward. The case of $f = \neg f'$ is less obvious, but since we have defined $WPC(s, f)$ as the weakest condition, $\sigma \not\models WPC(s, f)$ immediately implies $\sigma \models WPC(s, \neg f)$, and vice versa. Hence we have $WPC(s, \neg f) = \neg WPC(s, f)$.

Consider the case of $f = <a>_{\neg p} f'$. Suppose $\sigma$ is a value-assignment such that $(s, \sigma) \models <a>_{\neg p} f'$. From Definition 3.2, there must exist a concrete transition
sequence \((s, \sigma) \xrightarrow{t} (s, \sigma + t) \xrightarrow{a} (s', \sigma')\) such that \(t' \sim p\) and \((s', \sigma') \models f'.\)

Thus, the following conditions must also hold:

- some timed automaton transition \(s \xrightarrow{a,P,r} s'\) must exists.
- \(\sigma + t\) must satisfy both \(\text{Inv}(s)\) and \(P\) (Definition 2.2).
- \(\sigma'\) is a value-assignment \(\sigma + t\) whose values of the clocks in \(r\) are reset to zero, i.e. \(\sigma' = (\sigma + t)[r \rightarrow 0]\) (recall that \(\sigma + t\) is the same value-assignment as \(\sigma\) except all clock values are increased by \(t\), and \(\sigma[r \rightarrow 0]\) is the same as \(\sigma\) except all the clocks in \(r\) are reset to zero, as defined in Definition 2.2) and it satisfies \(\text{Inv}(s')\).
- \((s', \sigma') \models f',\) i.e. \(\sigma' \models WPC(s', f')\).

Hence, we obtain a necessary condition

"there exists some non-negative real-number \(t\) and some transition \(s \xrightarrow{a,P,r} s'\), such that \(\sigma \models (t \sim p)\), \(\sigma + t \models \text{Inv}(s) \land P\) and \((\sigma + t)[r \rightarrow 0] \models \text{Inv}(s') \land WPC(s', f')" 

for \(\sigma\) to make state \(s\) satisfy \(f\). We can rewrite \(\sigma + t \models \text{Inv}(s) \land P\) to the condition of \(\sigma\), such as \(\sigma \models (\text{Inv}(s) \land P)[C + t/C]\). By the same way, we can also rewrite \((\sigma + t)[r \rightarrow 0] \models \text{Inv}(s') \land WPC(s', f')\) as \(\sigma \models (\text{Inv}(s') \land WPC(s', f'))\) as
\( WPC(s', f')[C + t/C, 0/r] \). Therefore, the following condition holds:

\[
\sigma \models (0 \leq t \wedge t \sim p \wedge Inv(s) \wedge P \wedge (Inv(s') \wedge WPC(s', f'))[0/r])[C + t/C].
\]

Since it is sufficient that some non-negative real-number \( t \) and some transition \( s \xrightarrow{a_{P,r}} s' \) exist, we can weaken the above condition as:

\[
\sigma \models \exists t (0 \leq t \wedge t \sim p \wedge Inv(s) \wedge \bigvee_{i \in I(s, a)} \{ P_i \wedge (Inv(s_i) \wedge WPC(s_i, f'))[0/r]\})[C + t/C].
\]

where \( I(s, a) \) is a set of indices of transitions whose source node is \( s \) and action name is \( a \). We can easily prove that this is the weakest condition of \( \sigma \) such that \( (s, \sigma) \models f \), and \( t \) is some fresh variable which does not appear in either \( Inv(s), P_i, Inv(s_i) \text{ or } WPC(s_i, f') \).

Consider the case of \( f = f_1 E_{-p} f_2 \). Similar to above, suppose \( \sigma \) is a value-assignment such that \( f \). From Definition 3.2, there must exist some transition sequence \( (s, \sigma) = (s_1, \sigma_1) \xrightarrow{t_1} \cdots \xrightarrow{t_{k-1}} (s_k, \sigma_k) \) such that \( f_2 \), \( t \sim p \), and for any \( j (1 \leq j \leq k) \) and for any \( t_j (0 \leq t_j < t) \), \( (s_j, \sigma_j + t_j) \models f_1 \) holds. To obtain a recursive definition of \( WPC(s, f) \), we divide the premise of the above statement into two cases, \( k = 1 \) (\( f_2 \) holds at the current state \( s_1 \)) and \( k \geq 2 \) (\( f_2 \) holds at some future state \( s_k \)).

**[Case \( k = 1 \)]**: If \( k = 1 \), then there must exist a transition sequence \( (s, \sigma) \xrightarrow{t} (s, \sigma + t) \) such that \( (s, \sigma + t) \models f_2 \), \( t \sim p \) and for any \( t' (0 \leq t' < t) \), \( (s, \sigma + t) \models f_1 \) holds. Similar to the case of \( f = \langle a \rangle_{-p} f' \), the weakest condition of \( \sigma \) is obtained as follows:

\[
\sigma \models \exists t (0 \leq t \wedge t \sim p \wedge (Inv(s) \wedge WPC(s, f_2))[C + t/C] \wedge \forall t' ((0 \leq t' \wedge t' \leq t) \Rightarrow WPC(s, f_1)[C + t'/C])) \tag{1}
\]

**[Case \( k \geq 2 \)]**: If we assume \( k \geq 2 \), then there must exist a transition sequence \( (s, \sigma) = (s_1, \sigma_1) \xrightarrow{t_1} \cdots \xrightarrow{t_{k-1}} (s_k, \sigma_k) \) such that \( (s_2, \sigma_2) \models f_1 E_{-p} f_2 \) holds, and for any \( t_1 (0 \leq t_1 \leq t_2) \), \( (s_1, \sigma_1 + t_1) \models f_1 \) holds. Considering that there may exist multiple transitions \( s \xrightarrow{a_{P,r}} s_i \), the weakest condition of \( \sigma \) is obtained as follows:

\[
\sigma \models \exists t (0 \leq t \wedge (Inv(s) \wedge \bigvee_{i \in I(s)} \{ P_i \wedge WPC(s_i, f_1 E_{-p} f_2))[C + t/C] \wedge \forall t' ((0 \leq t' \wedge t' \leq t) \Rightarrow WPC(s, f_1)[C + t'/C])) \tag{2}
\]
where $I(s) \equiv \{i | s \xrightarrow{a_i, p_i} s_i\}$ is a set of indices of transitions whose source node is $s$.

Therefore, the general case is (1) or (2), that is,

$$\sigma \models \exists t[0 \leq t \land \forall t'[(0 \leq t' \land t' \leq t) \Rightarrow WPC(s, f_1)[C + t'/C]] \land (Inv(s) \land (t \sim p \land WPC(s, f_2) \lor \bigvee_{i \in I(s)} \{P_i \land WPC(s_i, f_1 EU_{\sim(p-t_i)} f_2))[C + t/C]\]

The case of $f = f_1 AU \sim p f_2$ is similar, and we omit the detailed description due to space limitation.

If the transition graph contains no loops, there are no cases that $WPC(s, f)$ is recursively called during the computation of $WPC(s, f)$ itself. Thus, the function call $WPC(s, f)$ is ensured to terminate. Hence, a recursive function $WPC(s, f)$ is an algorithm to obtain the parameter condition for DAG-formed models (i.e. parametric timed DAG-automata).

**Theorem 4.1** For every state $s$ of a parametric timed DAG-automaton $M$ and every RPCTL formula $f$, a recursive function $WPC(s, f)$ always terminates and returns a correct solution of a parameter condition derivation problem, i.e.:

$$\forall \sigma. [\sigma \models WPC(s, f) \text{ iff } (s, \sigma) \models f]. \quad \square$$

### 4.2. Case of Periodic Models

If parametric timed automata have some loops, the algorithm $WPC(s, f)$ in Theorem 4.1 may not terminate. In this section, we prove that if models are periodic timed automata, we have only to check a finite fragment of the computation tree to derive the weakest condition of parameters.

At first, we introduce the notion of unfolding. Replace all returning transitions of a periodic timed automaton (Fig. 6-(a)) with transitions to the special terminating state. We obtain the corresponding parametric timed DAG-automata (Fig. 6-(b)). Then, attach the copy of the corresponding parametric timed DAG-automaton to the special terminating state of itself. Finally, we have the parametric timed DAG-automaton which represents the first 2 periodic behavior (Fig. 6-(c)). We refer to such a model as a 2-*unfolding* of the periodic timed automaton. Similarly, we can also define a $k$-unfolding of a periodic timed automaton as the parametric timed DAG-automaton which represents the first $k$ periodic behavior.

For any RPCTL operators except ‘until’ operators $EU$ and $AU$ (and all the operators derived from $EU$ and $AU$ such as $AG$, $AF$, $EF$ and $EG$), the weakest condition of parameters are obtained by just checking the current state (and the
next state if it is ‘next’ operators) and checking for the \textit{proper} subformulas recursively. Since the size of each subformula is strictly decreasing, the recursion eventually terminates and we can obtain the result. However, it is not the case for the ‘until’ operators $EU$ and $AU$. The definition of $WPC(s, f_1 EU_{= p - 1} f_2)$ contains not only recursive calls for its proper subformulas $f_1$ and $f_2$, but also the recursive call for the formula $f_1 EU_{= (p - t)} f_2$, which is not a proper subformula. This means that it may execute the function call of the form $WPC(s_t, f_1 EU_{= (p - t_1 - \ldots - t_k)} f_2)$ forever if the model contains some loops. Our result is that without loss of generality, we have only to check 3-unfolding of periodic timed automata for each subformula including ‘until’ operators ($EU$ and $AU$).

Formally, it is proved by the following lemma:

\textbf{Lemma 4.1} For any concrete state $(s, \sigma)$ of periodic timed automata, the following condition holds:

$$(s, \sigma) \models f_1 EU_{= p} f_2 \text{ if and only if } (s, \sigma) \models f_1 EU_{= p'} f_2$$

where $T$ is the period of the periodic timed automata, $p' \overset{\text{def}}{=} p - m \times T$ and $m$ is the minimum nonnegative integer s.t. $p - m \times T < 3T$. The same condition also holds for $f_1 AU_{= p} f_2$.

\textbf{(proof)} Let $\alpha, \alpha'$, etc. denote finite execution paths such as $(s_1, \sigma_1) \xrightarrow{t_1} (s_1, \sigma_1 + t_1) \xrightarrow{a_1} (s_2, \sigma_2) \xrightarrow{t_2} \ldots \xrightarrow{a_{k-1}} (s_k, \sigma_k)$. Let $ET(\alpha)$ denote the execution time of the path $\alpha$, i.e., $ET(\alpha) = t_1 + \ldots + t_{k-1}$. Consider there exists a path $\alpha$ which begins with $(s, \sigma)$ such that $(s, \sigma) \models f_1 EU_{= p} f_2$ (which implies $ET(\alpha) \sim p$). The case of $0 \leq ET(\alpha) < 3T$ is trivial, since if $(s, \sigma) \models f_1 EU_{= p} f_2$ holds for $\alpha$, $p$ must be less than $3T$ for any case of $\models \in \{<, \leq, >, \geq, =\}$ which implies $p = p'$ from the definition of $p'$, and thus obviously $(s, \sigma) \models f_1 EU_{= p'} f_2$ holds, and vice versa. Consider the case of $ET(\alpha) \geq 3T$. As illustrated in Fig. 7, there must also exist a path $\alpha'$ containing at least one cycle, such that $0 \leq ET(\alpha') < 3T$ and $(s, \sigma) \models f_1 EU_{= p} f_2$ holds. $\alpha'$ is obtained by removing cycles beginning with the initial state from $\alpha$ until $ET(\alpha) < 3T$ holds, while leaving at least one cycle.\footnote{Note that in a periodic timed automaton, all clocks are initially zero and reset to zero when returned to the initial state, whereas any other variables (parameters) are never modified during execution. So, the possible behaviour from the initial state is always the same, no matter how it is reached. Thus, after removing cycles beginning with the initial state from the execution path $\alpha$, it is still an executable path. Moreover, $f_2$ holds at the last state of the path and along with the path, $f_1$ always holds until $f_2$ holds. Therefore, if $ET(\alpha) \sim p$ holds, then $f_1 EU_{= p} f_2$ still holds on the shortened path.} Obviously, the relation between $ET(\alpha)$ and $ET(\alpha')$ is $ET(\alpha') = ET(\alpha) - m \times T$, where $m$ is the number of the removed cycles. Therefore, $ET(\alpha) \sim p$ implies $ET(\alpha') \sim p'$ where $p' = p - m \times T$, and immediately we have $(s, \sigma) \models f_1 EU_{= p'} f_2$. Conversely, if there exists a path $\alpha'$ containing at least one cycle, such that $0 \leq ET(\alpha') < 3T$ and $(s, \sigma) \models f_1 EU_{= p'} f_2$,
there must also exist a path $\alpha$ such that $ET(\alpha) \geq 3T$ and $(s, \sigma) \models f_1 EU_p f_2$, as also illustrated in Fig. 7. In this case, $\alpha$ is obtained by duplicating intermediate cycles of $\alpha'$ ($\alpha'$ should have at least one cycle by the assumption) repeatedly until $ET(\alpha) \geq 3T$ and $ET(\alpha) \sim p$ hold. The case of $f_1 AU_p f_2$ is similar (we consider a path which violates $f_1 AU_p f_2$, instead), and we omit the detailed proof due to the space limitation.

We define another algorithm $WPC_3(s, f)$ instead of $WPC(s, f)$ for periodic timed automata. $WPC_3(s, f)$ is almost the same as $WPC(s, f)$, except that $WPC_3(s, f)$ derives the weakest parameter condition such that $s$ satisfies $f$ within 3 periods.

By Lemma 4.1, it is sufficient to consider finite paths whose execution time is at most $3T$ in order to derive the weakest condition of parameters. Therefore, we obtain the following main theorem:

**Theorem 4.2** For every state $s$ of a periodic timed automaton $M$ and every RPCTL formula $f$, a recursive function $WPC_3(s, f)$ always terminates and returns a correct solution of a parameter condition derivation problem, i.e.:

$$\forall \sigma([\sigma \models WPC_3(s, f) \text{ iff } (s, \sigma) \models f])$$

5. **COMPUTATIONAL COMPLEXITY**

In this section, we evaluate the computational complexity of the functions $WPC(s, f)$ and $WPC_3(s, f)$. Direct implementation of the recursive function $WPC(s, f)$ (and $WPC_3(s, f)$) may perform very inefficiently because of repetitive function calls of the same tuple of arguments $(s, f)$, which is redundant. Thus, we evaluate the complexity when we use a cache to avoid unnecessary computation. We assume that the function $WPC(s, f)$ has its own cache table, which has the entry $(s, f, P)$ if $WPC(s, f)$ has already been computed and the result is $P$. If such a function is called, the cache table is checked first, and if
its value has already been computed, the cached result is returned. Otherwise, the computation is performed, the result is registered into the cache, and the result is returned.

The complexity of the above algorithm is evaluated as follows. For simplicity, we here ignore the length of the resulting formulas (i.e. we ignore the cost to simplify and/or check satisfiability of the formulas). Under the assumption, time and space complexity coincides, that is, the time necessary to compute $WPC(s, f)$ is equal to the size of the cache table, which is equal to the maximum number of the different tuples of arguments with which $WPC(s, f)$ will be called. From the definition of $WPC(s, f)$, the number of the different tuples of arguments is bounded by $n \times (m_1 + m_2)$, where $n$ is the number of control states reachable from $s$, and $m_1$ ($m_2$) is the number of the different subformulas of $f$ (the number of the different derived formulas of the form $f_1 op_{\cdots} f_2$, $op \in \{ EU, AU \}$, respectively). $m_1$ is equal to the number of nodes of the syntax tree of $f$. Thus, $m_1$ is $O(m \log m)$, where $m$ is the length of $f$. $m_2$ is equal to $m_1 \times l$, where $l$ is the depth of the DAG, i.e. the length of the longest directed paths from the root to leaf nodes of the DAG (here we refer to a leaf node of the DAG as a node which has no outgoing edges). Obviously, $l$ does not exceed the number $n$ of nodes. Overall, the number of different tuples of arguments (= the size of cache table) is $O(n \times m \log m) = O(n^2 m \log m)$. Since we have only to perform at least one computation for each tuple of arguments, the number of all computation is also $O(n^2 m \log m)$. Therefore, the time and space complexity of $WPC(s, f)$ is $O(n^2 m \log m)$.

The complexity of $WPC^3(s, f)$ is equal to the $WPC(s, f)$ for 3-unfolding DAG model. Let $n$ denote the number of nodes of a periodic model. Then, from the definition of unfolding, the depth of the corresponding 3-unfolding is $3n$. Therefore, by replacing $n$ with $3n$, we also conclude that the time and space complexity of $WPC^3(s, f)$ is $O(n^2 m \log m)$.

**Theorem 5.1** The time and space complexity of $WPC(s, f)$ and $WPC^3(s, f)$ for a DAG/periodic model $M$ is $O(n^2 m \log m)$, where $n$ is the number of control states in $M$, and $m$ is the length of RPCTL formula $f$.

**6. CONCLUDING REMARKS**

In this paper, we propose a method to derive a parameter condition for a periodic timed automaton which satisfies a property written in a temporal logic RPCTL formula.

Although our method applies to only the restricted class of timed automata, many real-time applications such as audio/video streaming, time sharing task schedulers can be specified as periodic timed automata.

The future works are to develop and improve the efficiency of the implementation of the algorithm, to extend our algorithm to simplify the parameter
condition and to handle multiple periodic timed automata which run concurrently. We are also considering to extend our method to handle some internal state variables of finite domain to improve expressiveness of the model.

REFERENCES


Part Four

Process Algebra
This page intentionally left blank.
PAMR: A PROCESS ALGEBRA FOR THE MANAGEMENT OF RESOURCES IN CONCURRENT SYSTEMS*

Manuel Núñez and Ismael Rodríguez
Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid, Spain.
{mn,ir}@sip.ucm.es

Abstract In this paper we present a process algebra for the management of resources in concurrent systems. Our aim is to define a formal framework that can help in the task of specifying systems that depend, for their execution, on a set of resources that they use. Usually systems consist in a set of processes. In order to improve their performance, these processes will be able to exchange resources among them. In our language, processes will consist in a behavior (formalized as a LOTOS process) and in information about the resources that they own. Systems will be defined as the parallel composition of a set of processes. We will study some examples applying the features of PAMR. These examples will try to show the usefulness of our language for specifying and analyzing concurrent systems where resources play an important role.

Keywords: Extension of FDT’s; Semantical Foundations; Process Algebras.

1. INTRODUCTION

During the last two decades Process Algebras have been used to specify and verify different kinds of systems. Nevertheless, most process algebraic models lack the ability to appropriately express the relation between the behavior of processes and the resources that they use. Actually, resources are usually modeled as processes. In this paper we will present a process algebra to deal with systems where resources must be taken into consideration. In order to illustrate the kind of systems that we will deal with, let us introduce the following simple running example. Let us consider a system consisting in the parallel execution of \( n \) subsystems \( (P_1 \ldots P_n) \) and \( m \) different kinds of resources that these subsystems may use (let us suppose that the total quantity of the resource \( i \) is

*Research supported in part by the CICYT project TIC 2000-0701- C02-0 1.
equal to $x_i$). The performance of these subsystems depends on these resources (for example, the portion of memory used by each subsystem, time quantum of CPU, time quantum of access to the bus, etc). Each subsystem $P_j$ has an initial distribution of resources $x_1^j \ldots x_m^j$, that is, in the beginning, subsystem $j$ owns $x_i^j$ units of the resource $i$. Given the fact that the quantity of resources that subsystems own cannot be bigger than the total amount, we work under the constraint: $\forall 1 \leq i \leq m : \sum_j x_i^j \leq x_i$. Finally, let us suppose that subsystems have a preference on how they like resources. For example, suppose that a subsystem $P_{j_1}$ runs at the same speed if we replace one unit of the resource $i_1$ by four units of the resource $i_2$, while another subsystem $P_{j_2}$ runs at the same speed if one unit of the resource $i_1$ is replaced by two units of the resource $i_2$.

In particular, $P_{j_1}$ will perform better if we replace three of its units of $i_1$ by one additional unit of $i_2$.

In order to describe this kind of systems, we will introduce a language where the usual description of the behavior of processes is extended with additional information. In our language, a process will not only take into account its activity (that is, which actions can be performed at a given point of time) but it will also consider which resources can be used during its execution. Besides, subsystems may exchange resources with other subsystems. For instance, in the example given above, if $P_{j_1}$ gives to $P_{j_2}$ three units of $i_2$ and receives from $P_{j_2}$ one unit of $i_1$, then both subsystems run faster. So, our language will provide mechanisms to, starting with an initial distribution of resources, accomplish a better performance of the studied system.

**Example 1** Consider a system where two programs are running. Using operating systems terminology suppose that one of the programs is an I/O-bound process (e.g. a process printing a paper) while the other one is a CPU-bound process (e.g. a process compiling the $\LaTeX$ source of another paper). Even if the initial distribution would assign the same resources to both processes (in terms of CPU use and access to I/O), it is clear that both processes would perform better if they exchange their resources in the adequate way. That is, even though the I/O-bound process will keep some time quantum of CPU, it is willing to reduce it (with respect to the original distribution) in order to get better access to I/O, and the other way around for the CPU-bound process.

We will define our language in two steps. First, we consider a base language where we can specify the usual behavior of processes. For this purpose, we will use a LOTOS like language. As we already said, processes will also contain information about the available resources: The resources that they own, the resources that they need to perform actions, etc. In particular, there will be a function indicating their preferences on resources: A utility function. This function will compute values according to the actions that the process is able to perform immediately. For example, consider a program that initially behaves as
a CPU-bound process and after a while its behavior is as an I/O-bound process. This must be reflected in its utility function. In the beginning, it will prefer to keep a bigger time quantum of CPU, but in the end, its utility function will report bigger values if it keeps better access to I/O.

The second step consists in the combination of processes to build systems. A system will be the parallel composition of a set of processes. Processes will be able to communicate with each other by two kinds of operations. First, we will have the usual mechanism for synchronization in parallel operators, that is, there will be some actions that processes may perform asynchronously and some actions that they have to perform synchronously. More important, in order to improve their performances, processes will be able to exchange resources. Besides, harmful exchanges will be forbidden. For instance, it will not be allowed that \( P_j \) exchanges three units of \( i_2 \) by one unit of \( i_1 \) with \( P_j \), because both subsystems get worse, and thus the whole system deteriorates. We will consider a parallel operator where communications are not restricted to be exactly between two processes. For instance, we will allow that three processes synchronize in an action \( a \) or that four processes exchange, in one step, resources among them.

Regarding exchange of resources, we will consider two policies. The first one, that we call preserving utility, will allow exchanges of resources only in the case that at least one of the processes improves its situation, and no process gets worse after the exchange. For example, this policy would not allow \( P_j \) to trade with \( P_j \), one unit of \( i_1 \) by one unit of \( i_2 \), because even though \( P_j \) improves its situation we have that \( P_j \) gets worse. The second policy, that we call maximizing utility, will not be based on the particular situations of processes but in the situation of the system. In this case, exchanges will be allowed only if they improve the overall performance of the system. For example, this policy would allow the previous exchange only if the profit for \( P_j \) is bigger than the loss for \( P_j \). In fact, this policy is very similar to consider that processes are not the owner of resources. This is so because part of the resources of a process can be expropriated (as far as the whole system improves).

In terms of related work, our language is based on a previous proposal [12]. In this paper we have extended that language to get a more appropriate framework for the specification of the systems that we are dealing with. Nevertheless, the theoretical framework introduced in [12], where some concepts of microeconomic theory were used, can be inherited by adding some slight modifications. There are models to specify systems sharing resources (e.g. [5]), but in this case resources are just accessed, not traded; this access induces some delays in the behavior of processes. If we consider the maximizing utility policy, our proposal is somehow related to the ODP trading function [9]. Nevertheless, under a preserving utility policy, it is the case that a process only uses (and nobody else can use them) the resources that it owns. Besides, under this pol-
icy, trade permanently transfers the ownership of the traded resources. Finally, management of resources appears in fields like operating systems or concurrent programming. Resources are usually owned by a mediator which allows the processes to use them. So, given the fact that we separate the behavior of a process and how it manages its resources, we think that our language can be successfully used for the specification of this kind of systems.

The rest of the paper is organized as follows. In Section 2 we define our language in terms of processes and systems. In Section 3 we present two examples of the use of our language and we study the absence of deadlocks for the defined systems. Finally, in Section 4 we present our conclusions and some directions for further research. An extended version of this paper, where a bisimulation semantics is defined and more complex examples are presented, can be found in [13].

2. DEFINITION OF THE LANGUAGE

In this section we present PAMR as well as its operational semantics. First, we will introduce the notion of process. Then, we will say that a system is the parallel composition of several communicating processes. We will define the operational semantics of systems by means of three rules. The first rule will describe how processes exchange resources among them. Processes will exchange resources until no more useful exchanges are possible. Then, the last two rules define how systems perform actions (possibly by synchronizing among the corresponding processes). We will finish this section by presenting some useful results for verifying some properties in Section 3.

Before we present the formal definition of our language, we introduce some mathematical notation which will be used in the rest of the paper.

**Definition 1** We consider \( \mathbb{R}_+ = \{ x \in \mathbb{R} \mid x \geq 0 \} \). By abuse of notation, we will consider \( \frac{r}{0} = \infty \). Let \( r \in \mathbb{R}_+ \). Then, \( \text{trunc}(r) \) denotes the natural number resulting from discarding the decimal part of \( r \) (e.g. \( \text{trunc}(1.2) = 1 \)).

We will usually denote vectors in \( \mathbb{R}^n \) (for \( n \geq 2 \)) by \( \bar{x}, \bar{y}, \ldots \). Given \( \bar{x} \in \mathbb{R}^n \), \( x_i \) denotes its \( i \)-th component. We extend some usual arithmetic operations to vectors. Let \( \bar{x}, \bar{y} \in \mathbb{R}^m \). We define \( \bar{x} + \bar{y} = (x_1 + y_1, \ldots, x_n + y_n) \). We write \( \bar{x} \leq \bar{y} \) if for any \( 1 \leq i \leq n \) we have \( x_i \leq y_i \).

We will usually denote matrices in \( A_{n \times m} \) (for \( n, m \geq 2 \), and a set \( A \)) by calligraphic letters \( \varepsilon, \varepsilon_1, \ldots \). Let \( A \in A_{n \times m} \), and \( i, j \) be such that \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \). We will denote the \( n \times (i - 1) + j \)th component of \( A \) by \( A_{ij} \), that is, if we consider \( A \) as a matrix, this component corresponds to the element located at file \( i \) and column \( j \).

Given a set \( A \), \( P(A) \) denotes the set containing all the subsets of \( A \).
The behaviors of processes will be defined by means of a usual process algebra. In this case, we will consider a simple LOTOS like base language. First, we give an auxiliary definition introducing the sorts for actions.

**Definition 2** Let \( \textbf{Act} \) be the set of **visible** actions \((a, b, \ldots \text{ range over } \textbf{Act})\). Let \( \text{Act}_\tau \) be the set of **internal** actions \((\mu, \mu, \ldots \text{ range over } \textbf{Act} \cup \text{Act}_\tau)\). We suppose that \( \tau \in \text{Act}_\tau \), that there exists a bijection \( f : \text{Act} \to (\text{Act}_\tau - \{\tau\}) \), and \( \text{Act} \cap \text{Act}_\tau = \emptyset \). Given a visible action \( a \in \text{Act} \), we will denote \( f(a) \) by \( \tau_a \). We denote by \( \text{ACT} \) the **set of actions**, that is, \( \text{ACT} = \text{Act} \cup \text{Act}_\tau \). Finally, let \( \text{Id} \) be a set of **(basic) process identifiers**.

Let us note that we will have not only a unique internal action, but a whole set of them. In addition to the usual \( \tau \) action, we consider an internal action for each of the visible actions. For an external observer, all the internal actions will be equal. The difference among them comes from the fact that they will need different resources to be performed. So, if a process needs a set of resources \( \tilde{x} \) to perform a visible action \( a \), and this action is hidden, the resulting action, that is \( \tau_a \), will need the same amount of resources \( \tilde{x} \) to be performed. Sets of internal actions appear in other models for concurrent processes (for example, for I/O automata [10]).

**Definition 3** The set of **basic processes**, denoted by \( B \), is given by the BNF-expression \( B ::= \text{stop } \big| X :\!\!=\!\!= X \big| X :\!\!= \: \big| B \big| B + B \big| B \big| A B \big| \text{hide } A \big| B \big| X := B \big| X := B \), where \( \mu \in \text{ACT} \), \( A \subseteq \text{Act} \), and \( X \in \text{Id} \).

The operational semantics for the base language is given in Figure 1, and it is standard. Let us remind that \( B \{B'/X\} \) represents the replacement of the free occurrences of the variable \( X \) in \( B \) by the term \( B' \). Let us also remark that, in rule (HID2), the result of hiding a visible action \( a \) is not \( \tau_a \) but \( \tau_a \). The
following definition will be used later on. It computes the actions that a basic
process may perform immediately

**Definition 4**

Let $B \in B$. We define its *set of immediate actions*, denoted by $\text{Imm}(B)$, as the set $\text{Imm}(B) = \{ \mu \in \text{ACT} \mid \exists B' \in B : B \xrightarrow{\mu} B' \}$.

As we sketched in the introduction, a process will not only consist in a
behavior. On the contrary, a process will keep track of the resources assigned
to it and of some information relating resources and actions. Specifically, a
process $P$ will be defined as a tuple $(B, \bar{x}, u, n, c)$ where the intuitive meaning
of the components is:

- $B \in B$ is a **basic process** indicating the behavior of $P$.
- $\bar{x} \in \mathbb{R}_+^m$ indicates that $P$ owns $x_i$ units of the $i$-th resource, for any
  $1 \leq i \leq m$.
- $u$ is a function indicating *preferences* between *baskets* of resources. This
  function takes as parameters a set of actions (the actions that $B$ may
  perform immediately) and a set of resources, and returns a real number.
  For example, $u(A, \bar{x}) < u(A, \bar{y})$ means that if $\text{Imm}(B) = A$ then $P$
  would prefer to own the basket $\bar{y}$ to $\bar{x}$. So, processes will try to increase
  the value of $u$, given $A$, by exchanging resources with other processes.
- $n$ is a function relating *resources* and speed of execution. This will
  be modeled by a function taking as parameters an action and a set of
  resources: $n(a, \bar{x}) = \infty$ means that $a$ needs more than $\bar{x}$ to be performed;
  $n(a, \bar{x}) = r \in \mathbb{R}$ means that $a$ takes $r$ units of time to be performed
  if the process has the use of the resources $\bar{x}$ . A necessary condition for
  $P$ to perform $a$ is $n(a, \bar{x}) < \infty$. Let us remark that in some situations
  there will be a strong relation between the functions $u$ and $n$: Bigger
  utilities will be produced by bigger amounts of (or better) resources, and
  this will imply faster performance of actions. So, we may have relations
  like $n(a, \bar{x}) = \frac{K}{u(\{a\}, \bar{x})}$, for a given constant $K \in \mathbb{R}_+$. Also note that this
  is not always the situation. For instance, there can be actions that do not
  need any resources to be performed, actions that need only a strict subset
  of them, etc. That is why we have preferred to keep both functions.
- $c$ is a function indicating the *consumed* resources after performing an
  action. In some situations, the execution of an action needs to consume
  some resources (for example, writing in a bounded buffer); in other situ-
  ations, resources are created after performing actions (for example, a pop
  operation from a bounded stack). These situations will be modeled by a
  function that takes as inputs an action and a set of resources, and returns
  a set of resources. That is, $c(a, \bar{x}) = \bar{y}$ means that, after performing
a, the set of resources of the process varies from \( \bar{x} \) to \( \bar{y} \). For example, if \( \bar{z} = c(a, \bar{x}) - \bar{x} \) then \( z_i > 0 \) indicates that, after performing \( a \), the process has created \( z_i \) units of the \( i \)-th resource. A necessary condition for a process to perform an action \( a \) is \( c(a, \bar{x}) \geq 0 \). We do not allow debts because they could generate inconsistencies. For example, such debts could produce that two processes use simultaneously a printer.

**Definition 5** Let us consider that there exists a number \( m > 0 \) of different resources. We say that the tuple \( P = (B, \bar{x}, u, n, c) \) is a process if \( B \in B \) (the basic process defining the behavior of the process), \( \bar{x} = (x_1, \ldots, x_m) \in \mathbb{R}^m_+ \) (the amounts of resources owned by \( P \)), \( u : \mathcal{P}(\text{ACT}) \times \mathbb{R}^m_+ \rightarrow \mathbb{R} \) (the utility function), \( n : \text{ACT} \times \mathbb{R}^m_+ \rightarrow \mathbb{R}_+ \cup \{\infty\} \) (the necessity function), and finally \( c : \text{ACT} \times \mathbb{R}^m_+ \rightarrow \mathbb{R}^m_+ \) (consumption of resources function).

Given a process \( P_i \), we will usually consider \( P_i = (B_i, \bar{x}_i, u_i, n_i, c_i) \), that is, indices will denote the process to which \( B, \bar{x}, \ldots \) are related. □

The utility and necessity functions must fulfill some simple rational properties. We suppose that utility does not decrease if resources increase, that is, for any \( A \in P(\text{ACT}) \), and any \( 1 \leq i \leq m \), we have \( \frac{\Delta u(A, \bar{x})}{\Delta x_i} \geq 0 \). Given the fact that utility functions will be always applied to the initial actions of a behavior \( B \), we suppose \( u(\emptyset, \bar{x}) = 0 \). This property means that a deadlocked process does not need any resources, so they will be shared by the rest of the processes (because those resources do not add any utility to the owner). In microeconomic theory, there are other restrictions that are usually imposed (strict monotonicity, convexity, etc) but they are not needed in our current framework (see [11, 12] for more details). Regarding the necessity function, we will impose the condition that the function \( n \) is non-increasing with the number of resources, that is, for any \( 1 \leq i \leq m \) and \( \mu \in \text{ACT} \), we have \( \frac{\Delta n(\mu, \bar{x})}{\Delta x_i} \leq 0 \). Moreover, we will suppose that \( n(a, \bar{x}) = n(\tau_a, \bar{x}) \), that is, a visible action \( a \) needs the same resources that its internal counterpart \( \tau_a \). Finally, let us comment on a trick that we will intensively use in the examples of Section 3. There are many situations where resources cannot be split. For example, it makes no sense to consider that a process owns half of a printer. We will indicate that half of a printer is so useful as no printer at all by using utility functions like 

\[
 u(A, (x_1, \ldots, x_i, \ldots, x_n)) = f(A, x_1, \ldots, \text{trunc}(x_i), \ldots, x_n)).
\]

ure 1) to processes. The idea is that a process may perform a transition \( \frac{\mu}{\cdot} \) if its corresponding behavior so does, and the process has enough resources. Formally, if \( P = (B, \bar{x}, u, n, c) \) then we have the following rule:

\[
(\text{PRO}) \frac{B \xrightarrow{\mu} B' \land n(\mu, \bar{x}) < \infty \land c(\mu, \bar{x}) \geq \bar{0}}{P \xrightarrow{\mu} P'}
\]
where \( P' = (B',c(\mu,\bar{x}),u,n,c) \). Let us note that we have overloaded the symbol \( \rightarrow \): It is used both to denote transitions of behaviors and transitions of processes.

A system will be the parallel composition of a (finite, non-empty) collection of processes. Processes will communicate in two ways: Either by synchronizing in the execution of actions or by exchanging resources. Given the nature of the systems that we would like to describe, we have decided to use a relatively complex parallel operator. Specifically, our parallel operator is a simplification of [7] where \( m \) among \( n \) cooperation is not considered (similar proposals appear in [3,6]). The idea is that synchronizations are not restricted to be binary. The parallel operator will have a tuple of subsets of Act as parameter: \((A_1,\ldots,A_n)\). So, if we have the parallel composition of \( P_1,\ldots,P_n \), a process \( P_i \) will perform the actions in \( \text{Act} - A_i \) asynchronously; if \( a \in A_i \) then any \( P_j \) such that \( a \in A_j \) must synchronize with \( P_i \) in order to perform \( a \). Similarly, exchanges of resources may involve more than two processes.

**Definition 6** Let \( A_1,\ldots,A_n \subseteq \text{Act} \). A system \( S \) consists in the parallel composition of \( n \) processes \( P_1,\ldots,P_n \) synchronizing, respectively, in the set \( A_i \). We denote the system \( S \) by \( ||_n A_i P_i \). We denote the set of systems by \( S \).

From now on we will assume that systems are initially *compatible* with the available resources. That is, if we want to specify a system having the use of a vector of resources \( \bar{r} \) by \( S = ||_n A_i P_i \) then the sum of the total quantity of resources initially assigned to \( P_1,\ldots,P_n \) must be equal to \( \bar{r} \), that is, \( \sum_i \bar{x}_i = \bar{r} \). In the following we introduce the operational rules for systems. The first rule uses two predicates that will be defined in forthcoming definitions.

**Definition 7** (*The Exchange Rule*). Let \( S = ||_n A_i P_i \) be a system, where for any \( 1 \leq i \leq n \) we have \( P_i = (B_i,\bar{x}_i,u_i,n_i,c_i) \). The operational transitions denoting exchange of resources that \( S \) may perform are given by the rule:

\[
(\text{Par1}) \quad S \xrightarrow{\epsilon} ||_n A_i P_i' \quad \forall 1 \leq i \leq n: \quad P_i' = (B_i,\bar{x}_i - \sum_j \epsilon_{ij} + \sum_j \epsilon_{ji},u_i,n_i,c_i)
\]

where \( \mathcal{E} \in (\mathbb{R}^+)^{n \times n} \). We denote by \( \xrightarrow{\ast} \) the reflexive and transitive closure of \( \rightarrow \). We say that \( S \) is a *local equilibrium*, denoted by \( S \not\xrightarrow{\ast} S' \), if there do not exist \( S' \) and \( \epsilon \) such that \( S \xrightarrow{\mathcal{E}} S' \).

Intuitively, a transition \( ||_n A_i P_i \xrightarrow{\epsilon} ||_n A_i P_i' \) indicates that \( P_i \) gives to \( P_j \) the quantities of resources indicated by \( \epsilon_{ij} \) and receives from it the quantities given by \( \epsilon_{ji} \). The total amount of resources that \( P_i \) owns is equal to its original resources minus the resources that it gives plus the resources that it receives. As we will see later, processes will not be allowed to perform actions until the system reaches a local equilibrium. The idea is that an equilibrium represents a
good distribution of resources (because no more useful exchanges can be made) and so processes must be delayed until a local equilibrium is reached. If we allow a process $P_i$ to perform transitions before the system is an equilibrium (for example, as soon as $n_i(a, \bar{x}_i) < \infty$ and $c(a, \bar{z}) \geq \bar{0}$ hold) then the system will not be working at its best possible performance(s).

Now we will define the predicates valid$(S, \varepsilon)$ and allowed$(S, \varepsilon)$. The first predicate holds if the processes do not give more resources that the ones that they initially own, and the diagonal of the matrix is filled with $\bar{0}$.

**Definition 8** Let $S = \bigcup_{i=1}^{n} P_i$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \bar{x}_i, u_i, n_i, c_i)$. We say that $E \in (\mathbb{R}_{++}^n)^{n \times n}$ is a valid exchange matrix for $S$, denoted by valid$(S, \varepsilon)$, if for any $1 \leq i \leq n$ we have $\Sigma_j \varepsilon_{ij} \leq \bar{x}_i$ and $\varepsilon_{ii} = \bar{0}$.

Regarding the predicate allowed$(S, \varepsilon)$ we have several possibilities. This choice would depend on the kind of policy¹ that we want to implement. As we said in the introduction we consider two possible policies. The first policy, the preserving utility policy, will allow exchange of resources only if, after the exchange, at least one process improves and no process gets worse. Intuitively, processes are the owners of the resources and they will not give up them if they do not receive a compensation. The second policy, the maximizing utility policy, will allow exchanges if the overall situation of the system improves. In order to measure the improvement of the system, we consider the sum of the utilities of all the processes.² As we commented in the introduction of this paper, such a policy could be interpreted by thinking that processes are not the owners of resources because they can be expropriated without any compensation. Nevertheless, this policy is more efficient than a totally centralized system of resources delivery because it avoids the return and redelivery of the same resources to a process. In economic terms, the first of these policies can be seen as a kind of market economy while the second one could be interpreted as a kind of planned economy.

**Definition 9** Let $S = \bigcup_{i=1}^{n} P_i$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \bar{x}_i, u_i, n_i, c_i)$, and let $\varepsilon$ be a matrix such that valid$(S, \varepsilon)$.

The allowed$(S, \varepsilon)$ predicate under a Preserving Utility Policy is defined as $\forall 1 \leq i \leq n$ we have $u_i(\text{Imm}(B_i), \bar{x}_i) \leq u_i(\text{Imm}(B_i), \bar{x}_i - \sum_j \varepsilon_{ij} + \sum_j \varepsilon_{ji})$ and $\exists 1 \leq k \leq n : u_k(\text{Imm}(B_k), \bar{x}_k) < u_k(\text{Imm}(B_k), \bar{x}_k - \sum_j \varepsilon_{kj} + \sum_j \varepsilon_{jk})$.

The allowed$(S, \varepsilon)$ predicate under a Maximizing Utility Policy is defined as $\Sigma_i u_i(\text{Imm}(B_i), \bar{x}_i) < \Sigma_i u_i(\text{Imm}(B_i), \bar{x}_i - \sum_j \varepsilon_{ij} + \sum_j \varepsilon_{ji})$.

¹The choice of a good policy is not a trivial task. Actually, it is impossible to choose a perfect policy. This problem is related with the social welfare aggregator problem. Arrow's impossibility theorem shows that there does not exist such an aggregator fulfilling a certain set of desirable properties (see [1] for more details).

²Using microeconomics terminology, this is a Benthan social welfare aggregator.
From now on, if the results depend on the chosen policy, we will indicate under which one we are working. Once we have finished the definition of rule (Par1), let us remark that exchange of resources produces a lot of non-determinism. For example, consider the processes \( P_j \) and \( P_{j2} \) presented in the introduction. For any \( a > 0 \), and \( 2 \leq x \leq 4 \), exchanges where \( P_{j1} \) receives \( a \) units of the resource \( i_1 \) from \( P_{j2} \) and \( P_{j2} \) gives \( x \cdot a \) units of the resource \( i_2 \) from \( P_{j2} \) will be allowed.

**Definition 10** *(The Synchronization Rules).* Let \( S = \|A_i P_i \) be a system. The operational transitions denoting the actions that \( S \) may perform are given by the rules:

\[
\begin{align*}
& \text{(Par2)} \quad S \not\rightarrow \land P_j \xrightarrow{\mu} P_j' \land \mu \notin A_j \\
& \quad S \xrightarrow{\mu} \|\|_{i_1} P_i^{n'} \quad \forall 1 \leq i \leq n: P_i^{n'} = \begin{cases} P_j' & i = j \\ P_i & i \neq j \end{cases}
\end{align*}
\]

\[
\begin{align*}
& \text{(Par3)} \quad S \not\rightarrow \land P_j \xrightarrow{a} P_j' \land a \notin A_j \land \forall 1 \leq k \leq n: (a \in A_k) \Rightarrow (P_k \xrightarrow{a} P_k') \\
& \quad S \xrightarrow{a} \|\|_{i_1} P_i^{n'} \quad \forall 1 \leq i \leq n: P_i^{n'} = \begin{cases} P_j' & \text{if } a \in A_i \\ P_i & \text{if } a \notin A_i \end{cases}
\end{align*}
\]

Let us note that we have overloaded the symbol denoting transitions. The condition \( S \not\rightarrow \) indicates that the system has reached a local equilibrium. If \( S \) is not a local equilibrium, more useful exchanges can be made (i.e. the system may improve) and so actions should not be performed. Rule (Par2) is applied for interleaving actions. In this case, the only process that changes is the one involved in the transition. Rule (Par3) says that if a process \( P_j \) may perform an action \( a \) belonging to its synchronization set \( A_j \), then all the processes having \( a \) in their synchronization sets must also perform \( a \); all these processes will perform this action synchronously.

Finally, let us remark that our process algebra is a conservative extension of LOTOS: It is enough to consider that the functions \( u \) and \( n \) are always equal to 0 and that \( c(\mu, \vec{x}) = \vec{x} \), for any \( \mu \in \text{ACT} \).

### 2.1. Some Properties of the Language

In this section we study some properties that local equilibria fulfill according to the functions defining the involved processes. These properties will be used in the next section of this paper when analyzing the defined specifications. The first result says that any system may evolve into a local equilibrium.

**Lemma 1** For any system \( S \) there exists \( S' \) such that \( S \leadsto S' \not\rightarrow \). \( \Box \)
The proof of this result is immediate. Let us remark that if $S$ is already a local equilibrium then $S' = S$ (that is, there is a $\sim^*$ derivation with length equal to zero). The following result states that once a local equilibrium has been reached, no process will keep resources that do not add utility to it if there exists another process that uses the resource.

**Lemma 2** Let $S = \|P_i\|^n_1$. Let us suppose that there exist two processes $P_i, P_j$, $> 0$, and $1 \leq i \leq m$ such that for any $\bar{x}$ we have

- $u_i(\text{Im}(B_i), \bar{x}) = u_i(\text{Im}(B_i), (x_1, \ldots, x_r - \epsilon, \ldots, x_n))$, and
- $u_j(\text{Im}(B_j), \bar{x}) < u_j(\text{Im}(B_j), (x_1, \ldots, x_r + \epsilon, \ldots, x_n))$.

Under the previous conditions, for any $S'$ such that $S \sim^* S'$, we have that $x'_{ir} \leq x_{ir} - \epsilon$, where $x_{ir}$ denotes the amount of the resource $r$ owned by $P_i$ in $S$ (similar for $x'_{ir}$ and $S'$).

*Proof Sketch:* By contradiction. Suppose $x'_{ir} > x_{ir} - \epsilon$. We have that, under both policies, there exists an exchange where $P_i$ gives part of the resource $r$ to a process $P_k$ (there is at least a process increasing its utility by increasing the amount of $r$). So, $S'$ is not a local equilibrium.

An immediate corollary of the previous result is that if the condition holds for any $\epsilon$ such that $x_{ir} \geq \epsilon > 0$ then $x'_{ir} = 0$. Note that in the previous result is essential to suppose that another process increases its utility by increasing its amount of the resource $r$. The next result states that under a maximizing utility policy, all the local equilibria that can be reached from a system $S$ have the same *global* utility.

**Definition 11** Let $S = \|P_i\|^n_1$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \bar{x}_i, u_i, n_i, c_i)$. The total utility of $S$, denoted by $\text{total}(S)$, is defined as $\sum_i u_i(\text{Im}(B_i), \bar{x}_i)$.

**Theorem 1** Let $S = \|P_i\|^n_1$ be a system, where for any $1 \leq i \leq n$ we have $P_i = (B_i, \bar{x}_i, u_i, n_i, c_i)$. Let $S_1, S_2$ be systems such that $S \sim^* S_1 \not\sim^*$, and $S \sim^* S_2 \not\sim^*$. If the maximizing utility policy is being used then we have $\text{total}(S_1) = \text{total}(S_2)$.

*Proof:* Let $S_1 = \|P_{1i}\|^n_1$ and $S_2 = \|P_{2i}\|^n_1$, where for any $1 \leq i \leq n$ we consider $P_{1i} = (B_i, \bar{x}_1, u_i, n_i, c_i)$ and $P_{2i} = (B_i, \bar{x}_2, u_i, n_i, c_i)$. Suppose that $\text{total}(S_1) > \text{total}(S_2)$. Let $\epsilon$ be an exchange matrix such that for any $1 \leq i \leq n$ we have $\sum_i E_{ji} - E_{ij} = x_{1i} - x_{2i}$ and $\text{valid}(S_2, \epsilon)$. Given the fact that $\sum_i x_{1i} = \sum_i x_{2i}$, such a matrix fulfilling $\text{allowed}(S_2, \epsilon)$ always exists. Therefore, there exists $S'$ such that $S_2 \sim^* S'$, which represents a contradiction.
Note that different local equilibria may have different assignment of resources among processes; the previous result only assures that the sum of the utilities is the same, not that the composition of the baskets are equal. Note that this result does not hold under a preserving utility policy: Different equilibria do not have necessarily the same total utility.

3. EXAMPLES

In this section we will show how PAMR can be used to specify and analyze concurrent systems where resources play an important role. We will present two classical examples: The dining philosophers and consumers/producers. Besides, we will study the absence of deadlock in these systems.

In the following we assume that an undefined value of a function is set to an arbitrary value. Actually, these cases will not be possible because they will correspond with a set of actions (resp. with an action) not reachable by the corresponding processes.

3.1. The Dining Philosophers

In this classical problem, five (male) philosophers stay by a dining table, which contains five forks. We will consider that these are the resources of the system: fork1, . . . , fork5. These philosophers have only two tasks: To think and to eat. When a philosopher wants to eat, he must take both forks staying besides his dish (we suppose that philosopher i must take forks i and (i mod 5) + 1). The behavior of the philosophers can be described as:

\[
\text{Philosopher } i := \text{think}_i \; ; \; \text{eat}_i \; ; \; \text{Philosopher } i
\]

We suppose that the initial distribution of forks gives to philosopher i the i-th fork, that is, \(\hat{x}_i = (\delta_{i1}, \ldots , \delta_{i5})\) where \(\delta_{ij} = 1\) if \(i = j\) and \(\delta_{ij} = 0\) otherwise. Utility functions are defined as:

\[
u_i(\{\text{think}_i\}, \hat{x}) = 0 \quad \quad u_i(\{\text{eat}_i\}, \hat{x}) = \text{trunc}(\text{fork}_i) \cdot \text{trunc}(\text{fork}_{(i \mod 5) + 1})
\]

That is, if a philosopher wants to eat, he gets utility only if he owns both forks; otherwise, it will be the same to have one or none. Besides, a philosopher gets no utility by holding useless forks. We set \(u_i(\{\text{think}_i\}, \hat{x})\) to an arbitrary value because no resources are needed to think. The necessity functions are defined as follows:

\[
n_i(\text{think}_i, \hat{x}) = K_i \quad \quad n_i(\text{eat}_i, \hat{x}) = \frac{E_i}{u_i(\{\text{eat}_i\}, \hat{x})}
\]

In this case, the time that philosophers spend thinking may be different, but does not depend on the resources. Besides, we also assume that they eat at different speeds. Finally, no resources are consumed by performing actions, so
the consumption function is defined as \( c_i(a, \vec{x}) = \vec{x} \). In conclusion, the system can be defined as

\[
\text{Dining\_Philosophers} = \| A_i P_i \]

where for any \( 1 \leq i \leq 5 \), \( A_i = \emptyset \) and \( P_i = (\text{Philosopher}_i, \vec{x}_i, u_i, n_i, c_i) \).

The following result shows that this system cannot get deadlocked. As in the next example, the definition of the utility functions plays an important role in the absence of deadlocks.

**Lemma 3** (Absence of Deadlocks for Dining\_Philosophers). Consider a system \( S \) such that

\[
\text{Dining\_Philosophers} \leadsto^* S_1 \xrightarrow{a_1} S'_1 \leadsto^* S_2 \xrightarrow{a_n} S'_n \leadsto^* S
\]

If \( S \) is not a local equilibrium then there exist \( S' \), \( e \) such that \( S \xrightarrow{e} S' \); otherwise, there exist \( S' \), \( a \) such that \( S \xrightarrow{a} S' \).

**Proof:** The first case is trivial from Lemma 1. If a philosopher is willing to think, this action may be performed (no resources are needed). Suppose that all the philosophers desire to eat, and none of them can. This implies that all of them have zero utility. Regardless of the chosen policy, \( S \) is not a local equilibrium, because any exchange where some philosopher takes his two forks is allowed (note that fractions of forks will not be exchanged because they do not increase utility).

### 3.2. The Bounded-Buffer Producers/Consumers Problem

Consider \( n \) producers and \( m \) consumers. The former have access to a buffer where they can place their products; these products will be taken by the latter. The buffer is bounded: A consumer cannot get out a product if the buffer is empty and a producer cannot put in a product if the buffer is full. In order to avoid any damage in the structure, the buffer must be accessed preserving mutual exclusion. The behaviors of the involved processes are given by:

\[
\begin{align*}
\text{Producer} & := \text{produce} ; \text{enqueue}_1 ; \text{enqueue}_2 ; \text{Producer} \\
\text{Consumer} & := \text{dequeue}_1 ; \text{dequeue}_2 ; \text{consume} ; \text{Consumer}
\end{align*}
\]

In order to visualize how mutual exclusion is implemented, the \text{enqueue} and \text{dequeue} operations are split into two different steps. The initial resources of the system are: a unit of mutual exclusion, zero produced units, and \( p \) free places. We assume that these resources are randomly distributed among the processes in such a way that \( \sum_{i=1}^{n+m} x_i = (1, 0, p) \). We will define a unique utility function for all consumers and producers:
Let us note that the order structure of the buffer is not fully represented: The distinct products or distinct free places are not distinguished. Regarding the necessity function, it can be defined from the utility function as:

\[ n(a, \bar{x}) = \frac{K_3}{u(\{a\}, \bar{x})} \]

Let us remark that if a process does not own the necessary resources, its utility function will return 0, and so, the necessity function will be equal to infinite (let us remember that we consider \( \frac{K_3}{0} = \infty \)). In this example, the last two resources are created and consumed, so we have to define the value of the consumption function in the appropriate way:

\[
\begin{align*}
  c(\text{enqueue}_1, \bar{x}) &= (x_1, x_2, x_3 - 1) \\
  c(\text{enqueue}_2, \bar{x}) &= (x_1, x_2 - 1, x_3) \\
  c(\text{produce}, \bar{x}) &= c(\text{enqueue}_2, \bar{x}) = (x_1, x_2 + 1, x_3) \\
  c(\text{consume}, \bar{x}) &= c(\text{dequeue}_2, \bar{x}) = (x_1, x_2, x_3 + 1)
\end{align*}
\]

Finally, the system may be specified as:

\[
Bounded\_Buffer = ||^{A_i}_{n+m} P_i
\]

where, for any \( 1 \leq i \leq n \) we have \( P_i = (\text{Producer}, x_i, u, n, c) \), and for any \( n + 1 \leq j \leq n + m \) we have \( P_j = (\text{Consumer}, x_j, u, n, c) \). Besides, for any \( 1 \leq k \leq n + m \) we have \( A_k = \emptyset \).

One important property that we must have is \textit{mutual exclusion}. The next result states that accesses to the buffer are done by preserving this property: Once a process performs an action belonging to \{\text{enqueue}_1, \text{dequeue}_1\}, it is assured that this process will perform the corresponding second part of the performed action (\text{enqueue}_2 and \text{dequeue}_2 respectively) before any other process performs an action belonging to \{\text{enqueue}_1, \text{dequeue}_1\}.

\textbf{Lemma 4} Let \( S \) be a system such that

\[
Bounded\_Buffer \xrightarrow{a_1} S_1' \xrightarrow{a_2} S_2' \xrightarrow{a_3} \cdots \xrightarrow{a_n} S \not\sim
\]
Let us suppose $S \xrightarrow{a} S' \xrightarrow{=^*} S'' \not\xrightarrow{=^*}$, where $a \in \{\text{enqueue}_1, \text{dequeue}_1\}$ and $a$ has been performed by $P_j$ (that is, $P_j \xrightarrow{a} P'_j$). For any transition $S'' \xrightarrow{b} S'''$ such that $b \in \{\text{enqueue}_1, \text{dequeue}_1, \text{enqueue}_2, \text{dequeue}_2\}$ we have that if $a = \text{enqueue}_1$ then $b = \text{enqueue}_2$ (resp. if $a = \text{dequeue}_1$ then $b = \text{dequeue}_2$) and this transition has been performed by the evolution of $P'_j$ from $S'$ to $S''$.

Proof Sketch: Let us consider the producers case (for consumers is similar). Under the maximizing utility policy, when a producer performs its $\text{enqueue}_1$ action, for any reachable local equilibrium, the unit of mutual exclusion resource is given again to that producer because no other process would have more utility than this one by owning it (note that this producer will have utility 2). Under the preserving utility policy, when a producer performs an $\text{enqueue}_1$ action, the mutual exclusion resource cannot be taken by another process because the utility of the owner would decrease.

Lemma 5 (Absence of Deadlocks for Bounded_Buffer). Let us consider a system $S$ such that $Bounded_Buffer \xrightarrow{=^*} S_1 \xrightarrow{a_1} S'_1 \xrightarrow{=^*} S_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} S'_n \xrightarrow{=^*} S$. If $S$ is not a local equilibrium then there exist $S', \varepsilon$ such that $S \xrightarrow{\varepsilon} S'$; otherwise, there exist $S', a$ such that $S \xrightarrow{a} S'$.

Proof Sketch: If there are no products then any producer will be able to produce. If there are no free places, any consumer will be able to consume. In both policies, the mutual exclusion resource will be taken (and owned) by a process only if it makes that process to perform an action. Therefore, at least one process will get the resources it needs.

4. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a formalism to specify systems where resources can be exchanged among subsystems. These exchanges will improve the performance of the system. We have studied some (theoretical) properties of the language. Finally, we have specified and studied two examples where the characteristics of our language have been shown.

The referees of this paper have suggested new interesting directions of research. For example, necessity functions play no relevant role in the current theory. We would like to define a real-time extension of our language where these functions play a fundamental role: They will induce delays in the behavior of processes. These delays can be either deterministic or can be defined by means of a random variable. In the latter case, we will consider some of the current models of stochastic process algebras (e.g. [2,4, 8]). Besides, we will consider that the exchange of resources takes time. So, sequences of $\xrightarrow{=^*}$ transitions should be grouped. Another interesting point is that utility functions (as well as necessity and consumption functions) are static, that is, they do not change along the performance of the system. Once time is taken into
account, it is straightforward to extend the previous framework to consider dynamic functions. Finally, the relation between our framework and management of software projects should be explored. Indeed, the tasks of a project can be seen as processes while the members of the project can be seen as the resources used by the tasks.

Acknowledgments. We would like to thank the referees of this paper for the careful reading and the very interesting suggestions to improve the current framework. We would also like to thank the referees of [12] for pointing out some interesting directions which have been reflected in this paper. Finally, we would like to thank David de Frutos, Natalia López, Javier Rodríguez, and Fernando Rubio for helpful discussions on the topic of this paper.

REFERENCES

A SYMBOLIC SEMANTICS AND BISIMULATION FOR FULL LOTOS

Muffy Calder  
*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ*  
muffy@dcs.gla.ac.uk

Carron Shankland  
*Department of Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA*  
carron@cs.stir.ac.uk

Abstract

A *symbolic* semantics for Full LOTOS in terms of *symbolic transition systems* is defined; the semantics extends the (infinitely branching) standard semantics by giving meaning to data parameterised behaviours, and provides a finitely branching representation for behaviours. Symbolic bisimulation is defined.

This extends our previous work [14], making the definitions more amenable to automated reasoning and processes with recursion.

Keywords: LOTOS, symbolic transition systems, symbolic bisimulation.

1. INTRODUCTION

LOTOS [11] is a message passing process algebra which combines two orthogonal languages: a process language, known as Basic LOTOS, with features from both CSP and CCS, and the equational abstract data type language ACT ONE. LOTOS is an ISO standard [11] whose given semantics is in terms of structured labelled transition systems. In this semantics (referred to here as the *standard semantics*), each data variable in a process is instantiated by every possible value of its corresponding type, resulting in infinite transition systems (both in breadth and in depth).

This approach has several drawbacks. First, it is impossible to use standard (finite state) model-checking techniques over infinite transition systems; the usual solution is to restrict the underlying datatypes. Indeed, any kind of auto-
mated reasoning becomes difficult, if not impossible. Second, because the data values are embedded in the transitions, any uniformities in the actions of the processes are lost. For example, the process description may make it clear that a particular action happens when the value of some variable lies between 3 and 42 (say), but that information is much harder to extract from the labelled transition system directly, especially if there are an infinite number of branches. Finally, as a consequence of this approach it is not possible to reason about partial, or data parameterised, behaviour expressions. Our experiences with LOTOS applications (e.g. [15, 16]) indicate that this is highly desirable.

The advantage of the standard approach is that it easily accommodates multi-way synchronisation, i.e. associative synchronisation between two or more processes. Multi-way synchronisation has led to a particular constraint-oriented style of specification in the LOTOS community which is particularly useful when building a system by layering behaviour on a simple building block.

The problem we address here and in [14] is how to reason over potentially infinite LOTOS processes while retaining multi-way synchronisation and without restricting the datatypes. Since the addition of data to the language is the reason for the problem, some sort of separation of the concerns of data and processes seems appropriate. There are three kinds of solution to this problem. The first is to get rid of data altogether in a brute force manner [1] which changes the behaviour of the process. The second is to construct a process representation of the data type [2, 9]. This approach only converts data operations into process operations; the data values are still present therefore the process remains infinite branching. The third solution is to adopt a symbolic approach such as the symbolic semantics for message passing CCS in [10]. We drew on this approach for our earlier work [14] in defining a symbolic semantics and bisimulation for LOTOS. That definition did not deal effectively with recursive processes, and could not be easily implemented (since it required an infinite stock of new names). These problems are addressed here and the definitions considerably revised. The symbolic approach is also taken by Eertink [8]. While Eertink’s semantics achieves a separation of the concerns of data and process, without losing information, it is rather operational, concentrating on using the semantics in a simulation tool. There are no equivalences or preorder relations associated with the semantics.

Our motivation is to define an entire symbolic framework for reasoning about LOTOS specifications including a symbolic semantics, appropriate logics, relations and tools. It is important that this new framework preserves the distinguishing features of LOTOS (presented in Section 2). In this paper we present the foundation of the framework: symbolic transition systems (STSs) and the axioms and rules for generating an STS from a (possibly open) LOTOS behaviour expression (Section 3). In order to define relations over STSs we must first define the notion of substitution over STSs (Section 4). We fol-
low this with the definition of symbolic bisimulation (Section 5). Symbolic
bisimulation should not lead to processes being distinguished which are not
in the standard semantics; similarly processes which are distinguished under
the standard semantics are not identified in the symbolic semantics. We have
proven this to hold for unparameterised processes, but the proof is the sub-
ject of a different paper. Broadly, we follow the approach taken in [10] for
symbolic transition graphs and message passing CCS, but our approach differs
in several significant ways to accommodate the particular features of LOTOS.
Finally, we draw our conclusions and discuss future directions.

2. DISTINGUISHING FEATURES OF LOTOS

We assume the reader is familiar with the standard LOTOS syntax and se-
manitics [11]. An accessible tutorial to LOTOS is [12]. Here we present three
related features distinguishing LOTOS from most of the standard process al-
gebras, particularly message passing CCS: multi-way (broadcast) synchroni-
sation, value negotiation, and selection predicates. These make it non-trivial
to directly apply the notion of symbolic transition and bisimulation of [10].

Multi-way synchronisation means that when two actions synchronise, with
possibly some data exchange taking place, the resulting action may be involved
in further synchronisation. As a result of multi-way synchronisation, it makes
less sense in LOTOS to refer to “input” events and “output” events. In LOTOS
an event offers a single value or a set of values drawn from a particular sort;
these two cases are distinguished by the use of ! or ? respectively.

Multiway synchronisation is achieved in the underlying transition system by
encoding data into transitions in both ! and ? events. This can be seen clearly
by referring to the rules from the standard [11] for generating a transition sys-
tem from action prefix events. The rule for ! events is straightforward:

\[ g!E; P \rightarrow^v_P P \]

where \( E \) is a data expression (with no variables in the case of the standard
LOTOS semantics) and \( v = [E] \) (i.e. the equivalence class of \( E \)). Perhaps less
obvious is the rule for ? events:

\[ g?x:S; P \rightarrow^v_P P[v/x] \]

where \( v \) is a ground term of sort \( S \), and \( P[v/x] \) denotes the substitution of \( v \)
for \( x \) in \( P \). This rule gives us an axiom schema for each \(?x\) event. For example,
see Fig. 1; each possible value of Nat results in a transition.

Thus, ? event offers correspond to a (possibly infinite) choice over all values
of the data type. The binding of \( x \) is defined at this point, i.e. the semantics is
early. A late semantics (where binding of variables to values is delayed as long
as possible) has no counterpart in the (standard) concrete semantics. This is in
constraint to value-passing CCS where both kinds of semantics are possible.
Encoding of values in the transitions affects the rules for synchronised parallelism. Consider a rule for CCS style two-way synchronisation. We use LOTOS syntax here for comparison:

There is a single transition associated with the ? offer (labelled with \( x \)) and a clear indication that value passing is occurring: \( x \) gets bound to the value \( v \). However, this approach is clearly limited to two-way synchronisation because the transition label becomes the unobservable \( i \) action which may not synchronise with any other action.

Contrast this with the LOTOS approach to synchronisation:

Here, one of the transitions generated by the axiom schema for ? offers is chosen to match the transition generated by the ! offer (where \( v = [E] \)). That is, there may be lots of other potential transitions from the state \( g?x : S; P_2 \) labelled with \( g \) and some value, but only one which is labelled with the exact value \( v \). In fact, this is only one case of the rule for parallelism. In particular, the processes in the premise may be a further parallel combination, yielding multi-way synchronisation. Different combinations of offers are also permitted, i.e. ! and ? offers can synchronise in any combination. The most unusual case is value negotiation, which arises when \( g?x : S \) and \( g ? y : S \) synchronise, the result being that for every possible value, \( x \) and \( y \) are bound to the same value thereafter. Both values must have the same type.

Selection predicates may restrict data further. For example, transitions where \( x \geq 42 \) are denied by \( g ? x : S[x < 42] \). If the selection predicate evaluates to \textit{false}, the event itself is prevented from occurring. In the case of value negotiation if both ? offers are qualified by selection predicates, then the value must satisfy both predicates. An important distinction from other process algebras is that selection predicates can refer to data in the current event (typically the guards found in other process algebras refer to data from previous events).
So, to preserve multi-way synchronisation and selection predicates in LOTOS, we cannot simply employ the CCS approach to ? offers. We therefore define a new semantics based on symbolic transition systems.

3. SYMBOLIC TRANSITION SYSTEMS FOR LOTOS

3.1 Preliminaries

We assume a countable set of variables, Var, ranged over by $x$, $y$, etc., and a (possibly infinite) set of values, Val, ranged over by $v$. We also assume a set of data expressions, Exp, which includes Var and Val and is ranged over by $E$, and a set of Boolean expressions, BoolExp, ranged over by $b$, including the constants $tt$ (true) and $ff$ (false). We also assume that we have a set of gates, $G$, ranged over by $g$. The set of events, denoted Act, ranged over by $\alpha$, comprises SimpleEv and StructEv. The set of simple events, SimpleEv, ranged over by $a$, is defined as $G \cup \{i, \delta\}$. (Recall that in LOTOS $i$ is the internal event and $\delta$ is the exit event.) The set of structured events, StructEv, contains all gate-expression combinations $gE$, as well as all combinations $\delta E$.

We assume the existence of the flattening function of the standard semantics [11]. The flattening function ensures that the given specification adheres to the LOTOS syntax, but also removes all hierarchical structure, ensures uniqueness of variable names, and that all names and types used are previously defined. The resulting object is called a canonical LOTOS specification.

We follow several naming conventions of the standard, including the use of a variable name to stand for a more complex structure including information not just about the name of a variable, but also its type and scope.

Variables and Substitutions. Variables and substitutions are over data, and typed, although we do not make this explicit, as noted above.

We assume a set new-var of fresh variable names. Strictly speaking, any reference to this set requires a context, i.e. the variable names occurring so far. For simplicity, we will assume that this context can be inferred, as required.

A substitution is a partial function from Var to Var $\cup$ Val, written as $[z/x]$ where $z$ is substituted for $x$. A substitution is denoted $\sigma$, and the composition of two substitutions $\sigma_1$ and $\sigma_2$ is denoted $\sigma_1 \sigma_2$, where $\sigma_2$ has precedence.

Structured Events. Multiple data offers at a gate, e.g. $g!x!y?n : \text{Nat} ; P$, are allowed by LOTOS syntax. For simplicity in the following we will assume that only one event offer can occur at a gate. The obvious generalisation to lists of event offers can be easily made. The function name() : Act $\cup \{\delta, i\} \rightarrow G \cup \{\delta, i\}$ extracts the gate name from a structured event and is defined in [11].
Free and Bound Variables. The variables occurring in a data expression $E$ are given by $\text{vars}(E)$. A behaviour expression may contain free and bound data variables. The free variables of behaviour expressions, denoted $fv(P)$, are defined in Definition 5 of the Appendix.

Informally, free variables arise through usage in an expression where the variable name has been previously bound in one of several ways: as a formal process parameter, by a ? event, a let clause, or an enable ($\triangleright\triangleright$) with accept clause. For example, in $g?x; g!x; \text{exit}$, all occurrences of $x$ are bound, but in $g!x; \text{exit}$, $x$ is free.

3.2 Symbolic Transition Systems

Following [10], symbolic transition systems (STS) are transition systems separating data from process behaviour by making the data symbolic. We define an STS to be a labelled transition system with variables, both in states and transitions, and conditions which determine the validity of a transition.

Definition 1 (Symbolic Transition Systems)
A symbolic transition system consists of:

- A (nonempty) set of states.
  Each state $T$ is associated with a set of free variables, denoted $fv(T)$.
- A distinguished initial state, $T_0$.
- A set of transitions $T \xrightarrow{b, \alpha} T'$ such that $fv(T') \subseteq fv(T) \cup fv(\alpha)$ and $fv(b) \subseteq fv(T) \cup fv(\alpha)$ and $\#(fv(\alpha) - fv(T)) \leq 1$.

Following convention, we shall often identify an STS with its initial state. Since one possible interpretation of states is to view them as labelled by behaviour expressions, the set of free variables of an STS $T$, $fv(T)$, can be defined as $fv(P)$, where $P$ is the behaviour expression labelling $T$.

3.3 Intuition

We give a symbolic semantics for LOTOS by associating a symbolic transition system with each LOTOS behaviour expression $P$, written $\text{STS}(P)$. Before giving the axioms and rules for the symbolic semantics, we give an example illustrating the concrete (Fig. 2) and symbolic (Fig. 3) semantics for the behaviour expression

$$g?x:\text{Nat}[x < 10]; h?y:\text{Nat}; h!x; \text{stop}$$

In the standard semantics, query offers are instantiated by explicit data offers. Therefore, in Fig. 2, the ? offers correspond to either many or an infinite number of transitions, each labelled by a concrete offer.
In the symbolic semantics, open behaviour expressions label states (e.g. $h!x;\ stop$), and transitions offer variables. The range of permissible values for the variables is determined by the Boolean conditions. Whereas the system in Fig. 2 has infinite branching, the system in Fig. 3 has only finite branching.

3.4 Key Features

Many of the rules given in the following section are very similar to those in the LOTOS standard; only a simple notational change to make the transition a symbolic one is required. We note here the main departures from the standard semantics, and differences from [10] and [14]).

- The syntactic distinction between the two kinds of data offer, i.e. between ? and !, has been lost. That is, both are represented by a transition labelled by a gate, an expression (possibly a simple variable) and a Boolean condition. Each offer is a set of values constrained in some way – expressed both by the form of the expression and by the condition of the transition. The type of offer can be determined by examining the free variables of the associated states, i.e. if $fv(E) \subseteq fv(T)$ then this is a free variable (in a ! offer, introduced previously), and if $fv(E) \not\subseteq fv(T)$ then this is a new variable (in a ? offer, introduced in this transition).
Guarding, prefix and parallelism are the only rules which alter transition conditions, but unlike the LOTOS standard we do not evaluate those conditions while constructing the STS.

Transitions associated with ? events may introduce new variables, in order to avoid variable name capture. For example, the query variable in $g ? x : S; P ||| g ! x; Q$ needs to be renamed in order avoid capturing the free variable in the right hand side. New variables may be necessary even when every ? variable in a specification is unique. For example, when a process is invoked more than once, e.g. $P[g] ||| P[g]$ where $P[g] = g ? x : S; P'$, then one of the $x$ variables must be assigned a unique name to avoid confusion. We assume that we may perform alpha conversion (renaming of free variables) whenever necessary.

This is a major difference from our previous work [14], in which new variables were introduced at every data offer and synchronisation. This style of semantics gave a very clear distinction between information about a value (in the Boolean condition) and the value itself (denoted by a variable name), but would be difficult to implement since so many new names are required. Hence our revised version presented here.

Transitions may have conditions which are not satisfiable, and may arise through unsatisfiable guards in the LOTOS description, or through unsatisfiable combinations obtained through synchronisation.

### 3.5 Axioms and Rules of Transition

In this section we give the rules to generate $STS(P)$, the symbolic semantics of behaviour $P$, from a given canonical LOTOS behaviour expression $P$. Following the LOTOS standard the rules are grouped according to syntactic structures (shown in the boxes). Axioms show the transition and resulting state from a given portion of LOTOS syntax (on the left of the transition). Rules show the same, but with some preconditions given above the line. Note that relabelling is not part of standard LOTOS syntax, but is introduced in order to define process instantiation (as in the LOTOS standard).

We give here a complete set of rules for deriving a symbolic semantics for a LOTOS expression; however, the most interesting rules are those for prefix, exit, guard, and parallelism.

<table>
<thead>
<tr>
<th>Prefix Axioms</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{prefix axioms} & : \quad a; P \xrightarrow{tt} a \rightarrow P \\
& \quad g \ d; P \xrightarrow{tt} g E' = P \\
\end{align*}
\]

\[
E' = \begin{cases} 
E & \text{if } d = ! E \\
x & \text{if } d = ? x : 
\end{cases}
\]
\[ E' = \begin{cases} 
E & \text{if } d = !E \\
x & \text{if } d = ?x : S 
\end{cases} \]

exit axioms

\[ \text{exit} \xrightarrow{\Delta} \text{stop} \]

\[ \text{exit(ep)} \xrightarrow{\Delta E'} \text{stop} \]

\[ E' = \begin{cases} 
E & \text{if } ep = E \\
x & \text{if } E = \text{any } S \text{ where } x \in \text{new-var.} 
\end{cases} \]

let rule

\[ \frac{P[E/x] \xrightarrow{\alpha} P'}{\text{let } x : S = E \text{ in } P \xrightarrow{\alpha} P'} \]

choice range rules

\[ \frac{P[g_i/g] \xrightarrow{\alpha} P'}{\text{choice } g \text{ in } [g_1, \ldots, g_n] \text{ [ ] } P \xrightarrow{\alpha} P'} \]

for each \( g_i \in \{g_1, \ldots, g_n\} \)

\[ \frac{P \xrightarrow{\alpha} P'}{\text{choice } x : S \text{ [ ] } P \xrightarrow{\alpha} P'} \]

par rule

\[ \frac{P[g_1/g] \ op \ldots \ op \ P[g_n/g] \xrightarrow{\alpha} P'}{\text{par } g \text{ in } [g_1, \ldots, g_n] \text{ [ op ] } P \xrightarrow{\alpha} P'} \]

where \( op \) is one of the parallel operators, \( ||, |||, \) or \( ||[h_1, \ldots, h_m]|| \), for some gate names \( h_1, \ldots, h_m \).

hide rules

\[ \frac{P \xrightarrow{\alpha} P'}{\text{hide } g_1, \ldots, g_n \text{ in } P \xrightarrow{\alpha} \text{hide } g_1, \ldots, g_n \text{ in } P'} \]

if \( \text{name}(\alpha) \in \{g_1, \ldots, g_n\} \)

\[ \frac{P \xrightarrow{\alpha} P'}{\text{hide } g_1, \ldots, g_n \text{ in } P \xrightarrow{\alpha} \text{hide } g_1, \ldots, g_n \text{ in } P'} \]

if \( \text{name}(\alpha) \notin \{g_1, \ldots, g_n\} \)
accept rules

\[
P_1 \xrightarrow{b} \alpha \rightarrow P' \\
P_1 \triangleright accept x : S \text{ in } P_2 \xrightarrow{b} \alpha \rightarrow P'_1 \triangleright accept x : S \text{ in } P_2
\]

if \( \text{name}(\alpha) \neq \delta \)

\[
P_1 \xrightarrow{b} \delta \rightarrow P'_1
\]

\[
P_1 \triangleright accept x : S \text{ in } P_2 \xrightarrow{b} \delta \rightarrow P_2[E/x]
\]

Similarly for \( \triangleright \) with no data.

disable rules

\[
P_1 \xrightarrow{b} \alpha \rightarrow P'_1
\]

\[
P_1 \triangleright P_2 \xrightarrow{b} \alpha \rightarrow P'_1 \triangleright P_2
\]

if \( \text{name}(\alpha) \neq \delta \)

\[
P_1 \triangleright P_2 \xrightarrow{b} \alpha \rightarrow P'_1
\]

if \( \text{name}(\alpha) = \delta \)

\[
P_2 \xrightarrow{b} \alpha \rightarrow P'_2
\]

\[
P_1 \triangleright P_2 \xrightarrow{b} \alpha \rightarrow P'_2
\]

general parallelism rules (synchronising)

\[
P_1 \xrightarrow{b_1} g \rightarrow P'_1 \quad P_2 \xrightarrow{b_2} g \rightarrow P'_2
\]

\[
P_1[g_1, \ldots, g_n] \triangleright P_2[b_1 \wedge g = P'_1[g_1, \ldots, g_n] \triangleright P'_2
\]

where \( g \in \{ g_1, \ldots, g_n, \delta \} \)

\[
P_1 \xrightarrow{b_1} gE_1 \rightarrow P'_1 \quad P_2 \xrightarrow{b_2} gE_2 \rightarrow P'_2
\]

\[
P_1[g_1, \ldots, g_n] \triangleright P_2[b_1 \wedge b_2 \wedge E_1 = E_2 \wedge gE_1 = gE_2 \rightarrow P'_1[g_1, \ldots, g_n] \triangleright P'_2
\]

when \( \text{vars}(b_1 \cup E_1) \cap \text{vars}(b_2 \cup E_2) = \emptyset \).

general parallelism rules (not synchronising)

\[
P_1 \xrightarrow{b} \alpha \rightarrow P'_1
\]

\[
P_1[g_1, \ldots, g_n] \triangleright P_2[b \sigma \rightarrow P'_1[\sigma][g_1, \ldots, g_n] \triangleright P_2
\]

\( \text{name}(\alpha) \notin \{ g_1, \ldots, g_n, \delta \} \)

\( \sigma = \{ [z/x] \text{ if } \alpha = g \text{ and } x \in \text{vars}(P_2) \text{ where } z \in \text{new-var} \}
\]

\( [\ ] \text{ otherwise} \)

Similarly for \( P_2 \).
4. STATE EQUIVALENCE AND SUBSTITUTION

When constructing STSs from behaviour expressions, straightforward syntactic substitution on behaviour expressions was employed. Thus, state equivalence is defined purely by syntactic equivalence of LOTOS behaviours. However, in order to define equivalence relations, preorders, or logics over STSs and to ensure that cycles (such as might arise from recursive processes) are handled correctly, we must define substitution on STSs. This is new from [14].

In [10], this problem is solved by introducing the concept of a “term”: a node in a symbolic transition system paired with a substitution. The same solution can be adapted for LOTOS.
Formally, a *term* consists of an STS, $T$, paired with a substitution, $\sigma$ such that domain $\sigma \subseteq \text{fv}(T)$. We write this as $T\sigma$ to indicate that the substitution is not applied directly to $T$, and use $t$ and $u$ to range over terms.

Definition 2 gives the rules for transitions on terms, given the corresponding transitions on STSs. Only three cases are required: dataless transitions, transitions arising from $!$ offers, and transitions arising from $?$ offers. In all cases, $\sigma' = \text{fv}(T') \triangleleft \sigma$, that is, the restriction of $\sigma$ to include only domain elements in the set $\text{fv}(T')$. The definition of free variables is extended to terms in the obvious way. Terms, rather than STSs, are used as the basis for defining the bisimulation in the next section, and the logic in [5, 6].

**Definition 2 (Transitions between terms)**

\[
\begin{align*}
T \xrightarrow{b} T' & \quad \text{implies} \quad T_{\sigma} \xrightarrow{b\sigma} T'_{\sigma}, \\
T \xrightarrow{gE} T' & \quad \text{implies} \quad T_\sigma \xrightarrow{gE}\sigma T'_{\sigma}, \text{where } \text{fv}(E) \subseteq \text{fv}(T) \\
T \xrightarrow{g\sigma} T' & \quad \text{implies} \quad T_\sigma \xrightarrow{g\sigma[x/z]} T'_{\sigma[x/z]} \text{ where } x \notin \text{fv}(T), z \notin \text{fv}(T_\sigma)
\end{align*}
\]

We note that while the vast majority of LOTOS specifications give rise to finite STSs, infinite (depth) STSs are still possible, for (data) parameterised processes. For example, $P(x) = g!x; P(x+1)$ has an infinite (depth) STS. We do not consider the possibility of infinite depth a major drawback for our approach, but are beginning to investigate ways of dealing with it.

5. **SYMBOLIC BISIMULATION**

Standard bisimulation on transitions without data requires only the gate names to be the same. The obvious extension to transitions with data requires both gate and value to be matched exactly. In the symbolic world this is not appropriate. Consider, a symbolic transition stands for a set of concrete transitions. There are many ways to split this set into subsets; each characterisation yields a different symbolic representation. Therefore, in symbolic bisimulation, all gate names have to match exactly, but there need not be a one to one correspondence between values. A single transition in one process can be matched by one or more transitions in the other.

For example, consider the processes in Figure 4. The processes are clearly bisimilar, subject to the bisimilarity of $T'$ with $U1$ and $U2$, but one transition in $T$, $x \in A \cup \overline{A}$, matches two transitions in $U$, $x \in A$ or $x \notin A$.

This idea of partitioning data into different sets, according to some Boolean expressions, is the crux of the following definition of symbolic bisimulation. The use of the partition means that each bisimulation is a parameterised family of relations, where the parameters are the predicates. Whereas Hennessy and Lin [10] define both an *early* and a *late* bisimulation equivalence, only an *early* bisimulation is meaningful in our context (as explained in Section 2).
We give a definition of “layered” symbolic bisimulation, written $\sim_i^b$, where $i$ is the depth of bisimulation and $b$ is the initial context (usually simply $tt$).

We assume a function $new(t, u)$ which, given two terms $t$ and $u$, returns a variable which is not among the free variables of either $t$ or $u$.

**Definition 3 (Symbolic Bisimulation on terms)**

For all $b$, a Boolean expression, $t$ and $u$, terms:

1. $t \sim_0^b u$.
2. For all $n > 0$, $t \sim_{n+1}^b u$ iff
   
   (a) **(dataless case)**
   
   if $t$ has a transition $t \xrightarrow{b_i \alpha} t'$ then there is a finite set of Boolean conditions $B$ over $fv(t)$ such that $(b \land b_t) \Rightarrow \bigvee B$ and for each $b' \in B$ there is a transition $u \xrightarrow{b_{x \in A} \alpha} u'$ such that $b' \Rightarrow b_u$ and $t' \sim_{n}^b u'$.

   (b) **(data case, no new variable)**
   
   if $t$ has a transition $t \xrightarrow{b \land b_t \land z = E_t} t'$, where $fv(E_t) \subseteq fv(t)$, then there is a finite set of Boolean conditions $B$ over $fv(t) \cup \{z\}$ such that $(b \land b_t \land z = E_t) \Rightarrow \bigvee B$, where $z = new(t, u)$, and for each $b' \in B$ either
   
   there is a transition $u \xrightarrow{b \land b_{x \in A} \alpha} u'$, where $fv(E_u) \subseteq fv(u)$, and $b' \Rightarrow b_u$ and $b' \Rightarrow E_t = E_u$ and $t' \sim_{n}^b u'$

   or

   there is a transition $u \xrightarrow{b \land b_{x \in A} \alpha} u'$ such that $b' \Rightarrow b_u$ and $t' \sim_{n}^b u'$

   (c) **(data case, new variable)**
   
   if $t$ has a transition $t \xrightarrow{b \land z = E_t} t'$, where $z = new(t, u)$, then there is a finite set of Boolean conditions $B$ over $fv(t) \cup \{z\}$ such that $(b \land b_t) \Rightarrow \bigvee B$ and for each $b' \in B$ either
   
   there is a transition $u \xrightarrow{b \land b_{x \in A} \alpha} u'$, where $fv(E_u) \subseteq fv(u)$, and $b' \Rightarrow b_u$ and $b' \Rightarrow z = E_u$ and $t' \sim_{n}^b u'$

   or

   there is a transition $u \xrightarrow{b \land z = E_t} u'$ and $b' \Rightarrow b_u$ and $t' \sim_{n}^b u'$

(d), (e), (f) Symmetrically, the transitions of $u$ must be matched by $t$. 

---

**Figure 4. Bisimulation example**
We may be relating processes that are parameterised. Therefore, the free variable (parameter) must be matched accordingly.

**Definition 4** (\(\sim^b\) for parameterised processes)

If \(fv(t) = \{x\}\), \(fv(u) = \{y\}\), and \(z = new(t, u)\) then

\[
t \sim^b u \iff \forall z. t[z/z] \sim^b[z/z, z/y] u[z/y].
\]

We use \(\sim^b\) to denote the largest symbolic bisimulation, for a given \(b\).

The role of the partition is to provide a step between the Boolean conditions of term \(t\) and those of term \(u\). This can be clearly seen in case (a). The intuition for cases (b) and (c) of Definition 3 is as follows. For case (b) we assume that the data of the \(t\) transition is a value (expression). The role of the new variable \(z\) is to provide a common language for matching transitions. In the context of \(b'\), a member of the partition, the expression \(E_t\) can either be matched by a \(u\) transition with an equivalent value \(E_u\), or a \(u\) transition with a new variable \(z\).

The rules for transitions between terms (Definition 2) allow the new variable \(z\) to be used without explicit renaming at this point. The conditions for matching vary depending on what sort of \(u\) transition is matched. Essentially, if a new variable is matched then conditions relating to the data are captured exactly by the condition \(b_u\). If a data expression is matched then information about data is given both by \(b_u\) and the expression \(E_u\). Case (c) considers the situation in which the data of the \(t\) transition is a new variable \(z\).

The resulting bisimulation is a Boolean condition-indexed relation. So, in most cases, when \(t \sim^b u\), and \(t\) evolves to \(t'\), and \(u\) evolves to \(u'\), and \(t' \sim^{b'} u'\), then \(b'\) is a different condition to \(b\), i.e. different states in the symbolic transition systems will be related by different members of the family of relations. This is because, in a typical symbolic transition system, restrictions on data increase with depth.

We have no space here to include an example; see [7].

### 6. CONCLUSIONS AND FURTHER WORK

We have defined a symbolic semantics for LOTOS in terms of symbolic transition systems, and symbolic bisimulation over those transition systems. Broadly speaking, we have adopted the approach of [10]; however, the features of LOTOS, especially the need to accommodate multi-way synchronisation and the resulting model of value passing, mean that this is not a straightforward adaptation of the theory presented in [10].

Our symbolic approach eliminates infinite branching which has been a major source of difficulty in reasoning about LOTOS specifications. The inference rules for the semantics are relatively simple and intuitive, and the meaning of unparameterised processes is the same as in the standard semantics (although we did not show that here).
We have only considered strong bisimulation here, though other relations (e.g. weak bisimulation) can be defined. While we have a means of checking whether a given relation is a symbolic bisimulation we have not given here an effective method of constructing that relation. However, it is fairly easy to see that the partition has to be derived from (the cross product of) the conditions in each transition system; an algorithm has been developed and implemented in Haskell. Worst case complexity is exponential, but early recognition of zeros (in the Boolean expressions) can lessen the pain somewhat.

Related work includes the definition of a modal logic FULL [5] which is adequate with respect to a version of the bisimulation defined here [6]. That is, it distinguishes and identifies exactly the same processes as that bisimulation. Tools to support model checking of FULL with respect to LOTOS specifications (or STSs) are also being developed based on several different implementation paradigms [3, 4, 13]. Further work is to add a notion of state (in the imperative sense) to deal with infinite (depth) transition systems.

Acknowledgments

The authors would like to thank Savi Maharaj for useful input on the definition of bisimulation, and Ed Brinksma for many fruitful discussions on reasoning about LOTOS. Carron Shankland thanks the British Council, the Nuffield Foundation and the Engineering and Physical Sciences Research Council (under the project “Developing Implementation and Extending Theory: A Symbolic Approach to Reasoning about LOTOS”) for their support.

REFERENCES


**Appendix: Auxiliary Definitions**

**Definition 5 (Free Variables)**

The free variables occurring in an expression is $fv(E) = vars(E)$, where $vars(E)$ denotes the variables occurring in expression $E$. The free variables of behaviour expression $P$, $fv(P)$, is defined below. Similarly for the set of free variables of an action $\alpha$, $fv(\alpha)$.

- $fv(stop) = \{\}$
- $fv(exit) = \{\}$
- $fv(exit(x)) = \{x\}$
- $fv(P[g]) = \{\}$
- $fv(P[g_1, \ldots, g_n]) = \{g_1, \ldots, g_n\}$
- $fv($) = fv(P)$
- $fv(g?x:S[SP]; P) = \{vars(SP) \cup fv(P)\} \setminus \{x\}$
- $fv(g!x[SP]; P) = \{x\} \cup vars(SP) \cup fv(P)$
- $fv(SP) \rightarrow P) = vars(SP) \cup fv(P)$
- $fv(\text{let } x = E \text{ in } P) = vars(E) \cup (fv(P) \setminus \{x\})$
- $fv(hide \ g \ in \ P) = fv(P)$
- $fv(\text{}(P_1; \times; P_2)) = fv(P_1) \cup fv(P_2)$, where $\times = \{, [\ldots, \ldots, [g_1, \ldots, g_n], ||, ||\}$
- $fv(\text{}(P_1; \Rightarrow accept \ x \in P_2) = fv(P_1) \cup (fv(P_2) \setminus \{x\})$
- $fv(\text{}(choice \ g \ in \ [g_1, \ldots, g_n] \{P\}) = fv(P)$
- $fv(\text{}(choice \ x \ in \ T \{\}) = fv(P)$
- $fv(\text{}(par \ g \ in \ [g_1, \ldots, g_n] \\op \ P) = fv(P)$ where $\op$ is one of the parallel operators
IMPLEMENTING A MODAL LOGIC OVER DATA AND PROCESSES USING XTL

Jeremy Bryans  
*Department of Computing Science and Mathematics, University of Stirling, UK*  
jwb@cs.stir.ac.uk

Carron Shankland  
*Department of Computing Science and Mathematics, University of Stirling, UK*  
ces@cs.stir.ac.uk

**Abstract**  
The modal logic FULL is designed to capture strong bisimulation over early symbolic transition systems (STSs) for full LOTOS. It provides a compact way of expressing and verifying properties involving both data and transitions. In this paper we present a restricted prototype implementation of a model checker for LOTOS for queries written using the FULL logic. The model checker is developed within the CADP package using XTL.

**Keywords:** Modal Logic, Symbolic LOTOS, CADP, XTL

1. **INTRODUCTION**  
Model checking (11) has proved to be a valuable verification method in recent years, and the size of systems which can realistically be verified is constantly increasing. However, there is still much room for improvement, and many techniques have been proposed to increase the state space of verifiable systems, including symmetry (13), abstraction (10) and symbolic methods (6). A particular problem is the inclusion of data in systems, since this can often lead to infinite state space. Typically this is dealt with by restricting the size of the data type. An alternative approach is to deal with data symbolically (16; 9). The state space is reduced by grouping transitions according to the kind of data passed, or the properties of that data. Thus, instead of investigating a single transition for every data value, groups of transitions are formed and verification is performed at the level of the groups.
Previous work (7; 8) has established a formal framework for the symbolic interpretation of LOTOS behaviours. The formal description technique LOTOS (18) was chosen due to its popularity and applicability to a wide range of applications (e.g. protocols and services (1), distributed systems (22), and as a semantics for higher level languages such as feature descriptions (26) and use-case maps (2)). Further, there are well established tools such as CADP (14) for reasoning about LOTOS behaviours. LOTOS is particularly amenable to the symbolic approach because it allows the description of process flow of control as well as allowing data to be passed between processes. Such data can affect the process flow of control and therefore cannot simply be ignored in verification.

On top of the symbolic interpretation of LOTOS an equivalence relation (9) and a modal logic called FULL (7) have been defined. The logic provides a way of expressing properties involving both data and transitions. Currently tools are being developed to support reasoning in all parts of the framework, including model checkers for the logic (5; 24). As an interim step, a prototype of the model checker has been implemented within CADP using XTL (21), a functional-type programming language allowing description of computation over graphs. This approach has limitations. The underlying semantics of CADP is not symbolic therefore the prototype is necessarily limited. The logic remains unchanged syntactically, but its expressive power is reduced since only properties over finite data types can be expressed. So, here we do not exploit the main advantage of our symbolic framework, which is to represent infinitely branching systems by finitely branching ones. However, the advantage of the approach is integration with a range of verification tasks already implemented in CADP, and the ability to experiment with the logic at an early stage.

The purpose of this paper is to present the XTL implementation of FULL, illustrate the sorts of symbolic properties (albeit over processes which have a concrete representation in CADP) which can be expressed and verified automatically, and discuss the ongoing plans for symbolic reasoning about LOTOS behaviours. We begin by introducing the CADP toolkit, summarising the main capabilities of interest to us. In Section 3 we present an overview of the logic FULL. We assume the reader is familiar with Full LOTOS, or at least process algebra. A more detailed introduction may be found in (7), where we discuss further the effects on the logic of the restrictions imposed by CADP. The main section of the paper is devoted to the implementation of modal operators with data using XTL. Finally we evaluate the success of this experiment, and present directions of ongoing and future work.
2. CADP

The CAESAR/ALDEBARAN development package (CADP) is a versatile multi-functional tool offering many different formal verification techniques, from interactive simulation through to compositional verification based on refinement. It is based around a common format for explicit Labeled Transition Systems (LTSs), known as Binary Coded Graphs, or BCGs. In particular, it accepts Full LOTOS as an input formalism, and offers a model-checking algorithm for expressions written in eXecutable Temporal Logic, or XTL. The BCG basis of CADP is finite and therefore all processes handled by CADP must use finite datatypes of up to 256 values.

XTL is a functional-type programming language designed to allow an easy, compact representation of common temporal logic operators. They are evaluated over LTSs encoded as BCGs. XTL provides low level operators which access the states, labels, and transitions of the LTS, and also operators to determine the successors and predecessors of states and transitions. A number of modal logics (eg. HML (17) and CTL (11)) have already been successfully encoded within XTL. CADP is therefore an ideal tool within which to build a prototype model checker for FULL, although the finite basis of BCGs means that we will not be able to exploit the full expressive power of the logic. This is discussed in more detail in the next section.

3. THE LOGIC FULL

The FULL logic was designed as part of an ongoing research project (12) to develop a framework for reasoning about Full LOTOS, i.e. the processes and the data. In this section we present an informal introduction to the symbolic logic FULL.

The syntax of FULL is based on a variant of HML (25), with quantifiers over data added. It is made up of two parts. The first set of formulae, ranged over by $\Phi$, applies to closed terms. The second set, ranged over by $\Lambda$, applies to terms with a single free variable, as would arise from a LOTOS process with a single parameter. (The extension to multiple free variables is straightforward.)

**Definition 1 Syntax of FULL**

$$\Phi ::= \ b \mid \Phi_1 ~ \& ~ \Phi_2 \mid \Phi_1 ~ \vee \Phi_2 \mid [a] \Phi \mid \langle a \rangle \Phi$$

$$\mid \langle \exists y \ g \rangle \Phi \mid [\exists y \ g] \Phi \mid \langle \forall y \ g \rangle \Phi \mid [\forall y \ g] \Phi$$

$$\Lambda ::= \exists y. \Phi \mid \forall y. \Phi$$

where $b$ is some Boolean expression, $a \in G \cup \{i, \delta\}$, $g \in G \cup \{\delta\}$, $G$ is the set of gate names, $i$ is the internal event, $\delta$ is the exit event and $y$ denotes a variable name.

The simple event operators are taken from HML (25) and are well known.
\( \langle a \rangle \Phi \): it is possible to do an event (and then satisfy \( \Phi \)), and

\([a] \Phi\): after every event, \( \Phi \) is satisfied. (May hold vacuously.)

In this logic, the novel operators are the four quantified modal operators. They are formed as combinations of the modes \( \langle \rangle \) and \( \[ \] \) and the quantifiers \( \exists \) and \( \forall \): \( \langle \exists y \rangle \), \( \[ \exists y \] \), \( \langle \forall y \rangle \) and \( \[ \forall y \] \). For each of these combination operators the meaning is determined using the quantifier to range over the value \( y \) being passed, and the mode to refer to the gate \( g \) at which it is being passed. For simplicity, we assume that a single data value is passed at a gate. The extension to multiple data offers at a gate is straightforward.

To introduce and illustrate these operators, we consider the example of the process \( P \) in Figure 1, which is taken from the introductory paper (7). This illustrates a number of the capabilities of the FULL logic.

When encoding the process \( P \) within CADP, we used the library NUM10.lib, which gives the natural numbers from 1 to 10 and offers the operators \( \text{GT} \), \( \text{LT} \) and \( \text{EQ} \) (>, < and =). This allows the finite BCG representation of \( P \) given in Figure 2. As noted above, the data type may have up to 256 values, but 10 makes the example easier to present.

Informally, the formula \( \langle \exists y \rangle \Phi \) is satisfied by a process if it can perform some transition with data \( y \) at a \( G \) gate, and the subsequent process satisfies the formula \( \Phi \). For example, the formula \( \langle \exists y \rangle \langle \text{K} \rangle \text{tt} \) is satisfied by the process \( P \), because there are values of \( y \) (in particular: 4, 5 and 9) which can lead to a state satisfying \( \langle \text{K} \rangle \text{tt} \).

The formula \( \langle \forall y \rangle \Phi \) is satisfied by a process if all the values that \( y \) can take have at least one \( G \) transition that leads to a process that satisfies \( \Phi \). For example, the formula \( \langle \forall y \rangle \langle \text{H} \rangle \text{tt} \) is satisfied by the process \( P \), because every \( y \) value has a possible \( G \) transition leading to a state satisfying \( \langle \text{H} \rangle \text{tt} \).

The formula \( \[ \exists y \] \Phi \) is satisfied by a process if there is some value that \( y \) can take, and every time the process performs a \( G \) transition with that value, the subsequent state satisfies \( \Phi \). For example, the formula \( \[ \exists y \] \langle \text{H} \rangle \text{tt} \) is satisfied by the process \( P \), because if \( y = 1, 2, 3, 5, 6, 7, 8 \) or 10 then it is always possible to perform an \( H \) action afterwards.

The formula \( \[ \forall y \] \Phi \) is satisfied by a process if for every possible value of \( y \), every \( G \) transition leads to a state satisfying \( \Phi \). For example, the formula \( \[ \forall y \] \langle \text{H} \rangle \text{tt} \lor \langle \text{K} \rangle \text{tt} \) is satisfied by the process \( P \), because every possible transition leads to a state which can either perform an \( H \) or a \( K \) action.

The logic is presented more completely in (7), where it is defined formally in terms of symbolic transition systems as derived from LOTOS (9). In order to determine the meaning of the logic (normally defined over symbolic transition systems) over Binary Coded Graphs, an equivalent interpretation of the logic over labelled transition systems is used. (See Appendix.) There is a direct
process P [G,H,K] : exit :=
    G?x:Num10 [x LT 5]; H; exit
    [] G!4; K; exit
    [] G?x:Num10 [x EQ 5]; (H; exit ! K; exit)
    [] G!5; H; exit
    [] G?x:Num10 [x GT 5]; H; exit
    [] G!9; K; exit
endproc

Figure 1. process P

Figure 2. Binary Coded Graph for process P
mapping between BCGs and LTSs. The equivalence of the two definitions of the logic is shown in (4).

The expressivity of the logic may be considered to be parameterised by the number of values of the data types used. For this prototype, CADP restricts us to using data types with only 256 values, therefore the expressivity of the logic is clearly restricted, since when interpreted over a symbolic transition system an infinite number of data types may be considered.

In practice, this restriction may not affect the verification process. It may be that only 256 distinct values are required to distinguish processes and properties. If this were not a possibility, then CADP would be almost useless as a verification tool.

As soon as more than 256 values may be used in one branching point the logic of the prototype can no longer be used to distinguish processes which can be distinguished by FULL.

Another consideration is efficiency. A symbolic transition system allows many (possibility infinitely many) transitions to be represented finitely (possibility by just one transition). If a data variable can have 256 values, CADP will construct a BCG with 256 corresponding transitions, and all of these transitions must be explored when verifying properties. Therefore more work has to be done than in the symbolic case, in which the same transitions may be represented by just one transition.

4. IMPLEMENTING FULL WITHIN XTL

We show in this section how to implement the simple modal operators ($\langle g \rangle$ and $[g]$) and the quantified modal operators ($\exists y g$, $\exists^d y g$, $\forall y g$ and $\langle \forall y g \rangle$) within the XTL language. The boolean operators $\wedge$, $\vee$ and $\neg$ have direct translations within XTL and are not shown here.

In XTL, a formula is associated with the largest set of states which satisfy it. Since the BCGs are finite these sets of states are necessarily finite.

To illustrate some useful XTL constructs, we begin with the simple modal operators, involving no data.

4.1 Simple Modal Operators

The $\langle a \rangle \Phi$ operator has two parameters, the simple gate name $a$ and the formula $\Phi$. To represent the gate $a$ we use the labelset $A$, and the formula $\Phi$ is represented by the set of states $\Phi_{i}$. Any state $S$ will satisfy this operator, provided at least one of the outgoing edges of $S$ is labelled $a$, (i.e. $\exists T: \text{edge among out}(S) \text{ in } (\text{label}(T) \text{ among } A))$, and the resultant state of that edge satisfies the formula $\Phi$ (i.e. $(\text{target}(T) \text{ among } \Phi_{i})$).

We define $\langle a \rangle \Phi$ as $\text{DIA } (A, \Phi_{i})$ within XTL by

```
def DIA (A : labelset, Phi : stateset) : stateset =
```
\{ S : \text{state where} \\
\exists T : \text{edge among out (S) in} \\
\text{(label (T) among A) and (target (T) among Phi)} \\
\text{end_exists}\}
\text{end_def}

Note here that any labelset A and any stateset Phi will be acceptable to DIA, but that to remain true to the FULL definition, A should be a single label representing the simple event a, and the stateset Phi should represent the FULL formula \Phi. This stateset can be supplied as the result of a nested function. For example, the formula \langle a \rangle \langle b \rangle tt would be implemented as DIA(a, DIA(b, tt)), where tt represents all states that satisfy true, i.e. all states.

The \([a] \Phi \) operator has the same two parameters, the gate name a and the formula \Phi. A state S will satisfy this operator, provided for any appropriately labelled outgoing edges of S, the resultant state of that edge satisfies the formula \Phi. This may be vacuously true if no edges are appropriately labelled. The differences from the implementation of \langle a \rangle \Phi are therefore the use of forall instead of exists, and implies instead of and. Again, although BOX takes a set of labels A and a set of states Phi, A should be a single label representing the simple event a, and the stateset Phi should represent the FULL formula \Phi.

We define \([a] \Phi \) as BOX(A, Phi) within XTL by

\text{def BOX (A : labelset, Phi : stateset) : stateset =} \\
\{ S : \text{state where} \\
\forall T : \text{edge among out (S) in} \\
\text{((label (T) among A) implies} \\
\text{(target (T) among Phi))} \\
\text{end_forall} \}
\text{end_def}

4.2 Quantified Modal Operators

The quantified modal operators are the ones which allow us to refer to the data part of Full LOTOS.

To understand the implementation of a particular quantified modal operator, we begin by considering a general formula \text{OP}(y,g) \Phi, where \text{OP} is one of the four quantified modal operators, y is a variable of type ytype and g is a gate name, of type ytype. We deconstruct \Phi into a form \Phi_y \wedge \Phi_1, where \Phi_y contains all the restrictions on the variable y, and \Phi_1 contains none of them.

The interpretation of each operator over a BCG can be understood as matching a particular pattern in the matrix of reachable states, as shown below.

The matrix in Figure 3 is formed from the BCG (Figure 2) of process P, where the headers of the columns are the labels of the outgoing transitions
\[ \Phi_1 = \langle H \rangle tt \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>P2</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>F</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>P3</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>T</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>P4</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>T</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>P5</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>P6</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>F</td>
</tr>
</tbody>
</table>

*Figure 3. Matrix representing the process P and \langle H \rangle tt*

from the initial state, and the labels on the rows are all the reachable states from the initial state.

For a particular label and state, the relevant position in the matrix can have one of three labels:

- **T**: a transition marked with this label can reach this state, and \( \Phi_1 \) holds,
- **F**: a transition marked with this label can reach this state, but \( \Phi_1 \) does not hold, or
- **o**: a transition marked with this label cannot reach this state.

The pattern on the matrix is then determined solely by \( \Phi_1 \), i.e. the part of the formula which is not concerned with the variable \( y \). Similar matrices can be built for other process states and other FULL formulae \( \Phi \).

The formula \( \Phi_y \), which contains all the restrictions on \( y \), determines which columns need to be considered when evaluating the complete formula \( \Phi_y \land \Phi_1 \). The validity of different FULL operators can be related to different patterns within the matrix.

As an example, let us consider the formula \( (y = 4) \land \langle H \rangle tt \), so \( \Phi_y \) is \( (y = 4) \) and \( \Phi_1 \) is \( \langle H \rangle tt \). We will consider prefixing this by each of the four modal operators in turn.

- \( \langle \exists y \ G \rangle \Phi \) needs just **one place** in the matrix to be labelled \( T \). But since our example is \( (y = 4) \land \langle H \rangle tt \) then this place must be in the \( G!4 \) column, because in every other column the formula \( (y = 4) \) is not true.

- \( [\exists y \ G] \Phi \) needs **one column** in the matrix which contains only the letters \( T \) and \( o \) in order to hold. But our specific example restricts this to the \( G!4 \) column, and it therefore does not hold. If instead we had \( \Phi \) defined as \( (y = 5) \land \langle H \rangle tt \) the formula \( [\exists y \ G] \Phi \) would be true.

- \( \langle \forall y \ G \rangle \Phi \) needs **at least one \( T \) in every column** in order to hold. Allowing \( \phi \) to restrict \( y \), although syntactically possible, makes the whole formula
immediately false because the quantifier ranges over all possible values of \( y \). Clearly, formulae such as \( \langle \forall y \ G \rangle \ (y = 4) \) must be false, since \( y \) can have values other than 4. However, the formula \( \langle \forall y \ G \rangle \Phi_1 \) holds.

- \( \langle \forall y \ G \rangle \Phi \) needs no Fs anywhere in the table in order to hold. Again, restricting \( y \) in \( \Phi \) immediately forms a false formula. In this case, since the table contains Fs, no formula \( \langle \forall y \ G \rangle \Phi \) holds. But if we constructed the table \( \Phi_1 = \langle H \rangle \ tt \lor \langle K \rangle \ tt \) then \( \langle \forall y \ G \rangle \Phi_1 \) would hold.

**IMPLEMENTATION** In order to interpret \( OP(y, G) \Phi \) within XTL, we will break \( \Phi \) into \( \Phi_y \Lambda \Phi_1 \) as described above. There will be no other variables in \( \Phi \) because any variables will have been bound in a previous step.

Each of the implementations of the quantified modal operators will receive two parameters: a labelset \( A \) which (loosely) corresponds to the combination of the gate \( g \) and the formula \( \Phi_y \), and a stateset \( Phi \) which corresponds to the formula \( \Phi_1 \). The implementation will return the set of all states which satisfy the operator. A formula is satisfied by a process if the initial state is in this returned set.

Whereas for the simple modal operators we assumed the labelset was in fact a singleton (to conform with FULL syntax), for the quantified modal operators the labelset will typically not be a singleton. Although in FULL we write a symbolic expression such as \( \exists y G \Phi \) this has to be expanded by XTL into a concrete set of labels. We use the (built in) XTL macro \( \text{EVAL}_A(\ A \) to generate the labelset of the actions which satisfy the predicate \( A \). For example, all \( y \) actions possible at a \( G \) gate are therefore evaluated as \( \text{EVAL}_A(\ G?y : ytype \) , yielding the set \{"G!1", "G!2", "G!3", . . . , "G!10"\}. This labelset is then supplied to the implementation of a quantified modal formula.

This is the point at which the restrictions placed on our implementation of the FULL model checker by the BCG implementation become evident. If we were adhering strictly to the semantics as expressed in the Appendix, then the labelset could be infinite (ranging across all values of the data type). Using the BCG representation we quantify over finite types.

The labelset may also encode \( \Phi_y \). To restrict \( y \) to values which satisfy \( \Phi_y \) we use \( \text{EVAL}_A(\ G?y : ytype \ where \ \Phi_y ) \). The type \( ytype \) can be a simple XTL type, or one of the types created for the LOTOS process. If we wish to use a created type we need to be careful to provide C implementations for the constructors and operators within the type definition, and to reiterate these within the \texttt{.xtl} file. In our example the CAESAR compiler identifies the NUM10 values as integers, and we can use \texttt{integer} in the XTL queries. But if we used the strings \texttt{one}, \texttt{two}, . . . etc. instead, then the compiler would implement these as character strings, and we would be forced to provide a C function to convert these back to NUM10 values within the \texttt{.xtl} file.
(\exists y \, G) \Phi \quad \text{A state } S \text{ will satisfy this operator, provided there is some outgoing edge in } A \text{ which hits the target set } \Phi (\text{i.e. } \exists T : \text{edge among out} \, (S)). \text{In matrix terms, finding one } T \text{ anywhere would be sufficient, provided } A \text{ made no restrictions on } y. \text{Otherwise the } T \text{ would have to be in one of the columns to which } y \text{ was restricted. This additional constraint is implemented wholly by the use of the } \text{EVAL}_A(G?y:ytype \, \text{where } \Phi) \text{ expression in generating a restricted labelset. See Section 5.1 for an example of this sort of formula. We can define } (\exists y \, G) \Phi \text{ within XTL as}

\begin{verbatim}
def DIAEVAL (A : labelset, Phi : stateset) : stateset = 
  \{ S : state where 
      exists T : edge among out (S) in
      (if label (T) among A 
        then (target(T) among (Phi)) 
        else false 
      end_if) 
  end_exists 
end_def
\end{verbatim}

The XTL definition above is a reasonably direct translation of the semantics of \( (\exists y \, G)\Phi \) as given in the Appendix. The if statement corresponds to finding one \( T \) in the matrix (with the right sort of label).

(\exists y \, G) \Phi \quad \text{A state } S \text{ will satisfy this operator, provided there is some label } L \text{ in } A \text{ (exists L : label among A)} \text{ such that every outgoing edge labelled with } L \text{ hits the target set } \Phi \text{ (the forall expression with nested if). That is, there is a whole column in the matrix containing only the letters } T \text{ and } o. \text{The else true implements the fact that this operator can be vacuously true if there is a label } L \text{ in } A \text{ which is not assigned to any outgoing edge, i.e. the corresponding column contains only the letter } o. \text{We define } (\exists y \, G)\Phi \text{ within XTL as}

\begin{verbatim}
def BOXEVAL (A : labelset, Phi : stateset) : stateset = 
  \{ S : state where 
      exists L : label among A in
      (forall T : edge among out (S) in
        if label (T) = L 
        then target (T) among (Phi) 
        else true 
      end_if 
      end_forall) 
  end_exists 
end_def
\end{verbatim}
It is harder to see a correspondence between the semantics of this operator as expressed in the Appendix and the XTL implementation above. The finite BCG semantics means that we effectively translate an expression of the form \( \exists y G \Phi \) into \( g_1 \Phi \lor g_2 \Phi \lor \ldots \lor g_{10} \Phi \), where each value of the data type yields a corresponding hardwired \([\ ]\) operator.

\((\forall y G) \Phi \) To implement the universally quantified operators we rely on the duality of \( \exists \) and \( \forall \), and \( \langle \ \rangle \) and \( [\ ]\), as pointed out in (7). In XTL, \( \text{not} (\Phi) \) is the complement of the set \( \Phi \). We define \( \langle \forall y \rangle \) as

```plaintext
def DIAAVAL (A : labelset, Phi : stateset) : stateset =
    not (BOXEVAL (A, not (Phi)))
end_def
```

taking advantage of the fact that \( \text{neg}(\langle \forall y G \rangle \Phi) \) is \([\exists y G] \text{neg}(\Phi)\).

\([\forall y G] \Phi \) We define \([\forall y G] \Phi \) as

```plaintext
def BOXAVAL (A : labelset, Phi : stateset) : stateset =
    not (DIAEVAL (A, not (Phi)))
end_def
```

taking advantage of the fact that \( \text{neg}([\forall y G] \Phi) \) is \( \langle \exists y G \rangle \text{neg}(\Phi)\).

5. THE MODEL CHECKER

To model check a LOTOS process within CADP using the FULL logic, we use the macro SAT, defined as

```plaintext
macro SAT (Phi) =
    PRINT_FORM( if Phi includes init then (TRUE) else (FALSE) end_if)
end_macro
```

This accepts a formula \( \Phi \) and prints TRUE if the initial state of the process is included in the stateset representing \( \Phi \), and FALSE otherwise.

We can then model check a process using

```
$xtl test.xtl process.bcg
```

The file process.bcg is the output from the caesar.adt and caesar components of the CADP tool. These take a LOTOS specification and produces a Binary Coded Graph (BCG). The file test.xtl contains a series of SAT commands. For convenience, SAT commands can also take a text parameter, which is returned with the model checker output. This is optional, but such comments can serve as a reminder of the formula being tested.
5.1 Examples

Here we use again the process $P$ from Section 3.

1. The query *is it possible for process $P$ to perform a $G!4$ action?* which is captured as $P \models (\exists y G)(y = 4)$ using FULL, is written

\[
\text{SAT( "<E y:G>(y=4)tt: ",}
\]
\[
(\text{DIAEVAL(EVAL_A(G?y:integer where y=4),TRUE)))}
\]

and because $(\text{EVAL_A(G?y:integer where y=4)})$ evaluates to "$G !4$" the formula becomes

\[
\text{if DIAEVAL{"G !4"}, TRUE) includes INIT then TRUE else FALSE}
\]
\[
\text{end_if}
\]

and for $\text{DIAEVAL \{"G !4\"}, \text{TRUE})$ to be true for process $P$, the initial state simply needs one outgoing $G!4$ action. This is clearly true (see Figures 2 and 3), so the output is

\[
<E y:G>(y=4)tt: \text{TRUE}
\]

2. The query *for all values $y$, is it possible to do a $G!y$ action followed by an $H$ action?* which is written as $P \models (\forall y G)\langle H \rangle tt$ using FULL, is expressed as

\[
\text{SAT( "<V y:G><H>tt: ",}
\]
\[
(\text{DIAVAL(EVAL_A(G?y:integer),}
\]
\[
(\text{DIA(EVAL_A(H),TRUE))))})}
\]

The parameter $(\text{EVAL_A (G?y:integer})$ evaluates to the set

\{'$G !1$', '$G !2$', '$G !3$', '$G !4$', '$G !5$',
'$G !6$', '$G !7$', '$G !8$', '$G !9$', '$G !10$'}

the restriction to ten elements coming from the process $P$, which is limited to the library NUM10.lib. This means every transition possible for $P$ must be considered. The operator $(\forall y G)$ requires that each label is on a transition leading to at least one state where the formula $\Phi$ is true. This is true here (see Figure 3, one $T$ in every column), and so the output is

\[
<V y:G><H>tt: \text{TRUE}
\]
3 The query for all values y, after any G!y action is it possible to do an H action? is written within FULL as $P \models [\forall y G] \langle H \rangle tt$ and is expressed as

$$\text{SAT( } \exists y [G] <H>tt: \text{,}
\left( \text{BOXAVAL(} \text{EVAL}_A(G?y:integer),}
\left( \text{DIA(} \text{EVAL}_A(H), \text{TRUE)}\right)\right)\right)$$

The parameter $\text{EVAL}_A(G?y:integer)$ generates the full list of possible transitions again, but this time the query fails, because, for example, it is possible to take a transition labelled $G!4$ to a state which only permits a $K$ action. It doesn’t matter that there is a successful $G!4$ transition which can subsequently perform an $H$ action.

The reasonably direct translation from FULL to XTL can be seen from these simple examples. The only tricky part is the need to provide requirements on a value $\Phi_y$ to the point at which that value is introduced.

6. CONCLUSIONS AND FURTHER WORK

We have successfully implemented a model checker for the FULL logic over LOTOS (with data), and we are planning to extend this work in a number of directions.

We have modelled the Remote Procedure Call case study (3; 15) using LOTOS within CADP, and performed some simple queries on it using FULL. We have yet to fully explore this example.

The FULL logic cannot as yet express mu-calculus type queries (20) with infinitely repeating patterns, but we are currently working on extending the logic in this direction. Further, for ease of presentation, the FULL logic is currently limited to single data values being passed at gates, and for practical use it would be preferable to allow multiple data values. For example, if $G?y1:bool?y2:integer$ was a gate, we would like to be able to identify all labels that matched $G!true?y2:integer$. Implementing the mu-calculus operators in XTL should be straightforward, but currently expressing pattern matching queries within XTL is not feasible.

The CADP toolkit provides a very accessible way of building a prototype model checker, and we are very pleased with the results obtained. However the BCG graphs are only ever finitely branching, because the LOTOS processes are restricted by the datatypes used. In order to reason about symbolic aspects of FULL allowing infinite data types we are considering two other approaches: theorem proving (using the Ergo tool) (24) and rewriting logic (5). Much interesting work on combining theorem proving and model checking is available (23; 19), and rewriting logic, while a comparatively recent technique,
has already been successfully used to implement a number of simple model
checkers (27). These techniques also have the advantage of allowing integra-
tion of reasoning about data and reasoning about processes.

Acknowledgments

Thanks to Radu Mateescu for the help provided for our work with CADP. The authors would like to thank the Engineering and Physical Sciences Re-
search Council for supporting this research under the project award “Develop-
ing Implementation and Extending Theory: A Symbolic Approach to Reason-
ing about LOTOS”.

REFERENCES


Appendix: The FULL logic definition

We give here the semantics of the logic, defined inductively over concrete labelled transition systems. It is assumed that \( \text{vars}(\Phi) \cap \text{vars}(t) = \{\} \), i.e. the variables of the transition system and the variables of the logical formula are disjoint.

First we define \( t \models \Phi \), denoting that a closed term \( t \) satisfies a closed modal formula \( \Phi \). The relation is defined inductively over the syntax of the logic by the equations of Definition 2.

The last two rules of the definition where \( t \) is an open term relate the (single) free variable of the parameterised transition system to the (single) free variable of a quantified logical property. The remaining part of the property is evaluated using the earlier \( t \models \Phi \) definitions.

**Definition 2** Semantics of FULL over a labelled transition system

Given any closed term \( t \), the semantics of \( t \models \Phi \) is given by:

\[
\begin{align*}
t \models b & \Rightarrow \ b \equiv tt \\
t \models \Phi_1 \land \Phi_2 & \Rightarrow t \models \Phi_1 \text{ and } t \models \Phi_2 \\
t \models \Phi_1 \lor \Phi_2 & \Rightarrow t \models \Phi_1 \text{ or } t \models \Phi_2 \\
t \models \langle a \rangle \Phi & \Rightarrow \text{there is a } t' \text{ s.t. } t \xrightarrow{a} t' \text{ and } t' \models \Phi \\
t \models [a] \Phi & \Rightarrow \text{whenever } t \xrightarrow{a} t' \text{ then } t' \models \Phi \\
t \models \langle \exists x \rangle \Phi & \Rightarrow \text{for some value } v, \text{for some } t', t \xrightarrow{\sigma^v} t' \text{ and } t' \models \Phi[v/x] \\
t \models \langle \forall x \rangle \Phi & \Rightarrow \text{for all values } v, \text{for some } t', t \xrightarrow{\sigma^v} t' \text{ and } t' \models \Phi[v/x] \\
t \models [\exists x] \Phi & \Rightarrow \text{for some value } v, \text{ whenever } t \xrightarrow{\sigma^v} t' \text{ then } t' \models \Phi[v/x] \\
t \models [\forall x] \Phi & \Rightarrow \text{for all values } v, \text{ whenever } t \xrightarrow{\sigma^v} t' \text{ then } t' \models \Phi[v/x]
\end{align*}
\]

Given any term \( t \) with one free variable \( z \) the semantics of an open formula, \( t \models \Lambda \), is given by:

\[
\begin{align*}
\models \exists x. \Phi & \Rightarrow \text{for some value } v, t[v/x] \models \Phi[v/x] \\
\models \forall x. \Phi & \Rightarrow \text{for all values } v, t[v/x] \models \Phi[v/x]
\end{align*}
\]

For a more complete explanation of the logic, consult [7].
Part Five

Applications of Verification
This page intentionally left blank.
FORMAL VERIFICATION OF PEEPHOLE OPTIMIZATIONS IN ASYNCHRONOUS CIRCUITS

Xiaohua Kong, Radu Negulescu
Montreal, Quebec, Canada H3A 2A7
Department of Electrical and Computer Engineering, McGill University
E-mail: {kong, radu}@macs.ece.mcgill.ca

Abstract
This paper proposes and applies novel techniques for formal verification of peephole optimizations in asynchronous circuits. We verify whether locally optimized modules can replace parts of an existing circuit under assumptions regarding the operation of the optimized modules in context. A verification rule related to assume-guarantee and hierarchical verification is presented, using relative timing constraints as optimization assumptions. We present the verification of speed-optimizations in an asynchronous arbiter as a case study.

Keywords: Peephole optimization, formal verification, peephole rule, relative timing, process spaces

1. INTRODUCTION

The dramatic increase of integrated circuit complexity raises the need for guarantees of correctness early in the design flow, instead of leaving such guarantees to testing after the design is completed. Presently, simulation is widely used to provide guarantees of correctness of designs. Unfortunately, simulation typically covers only a small subset of system behaviors, especially for asynchronous circuits, where several interleavings of signal transitions must be taken into account. In contrast, formal verification checks the correctness of a system under all possible signal interleavings. In this paper, we verify whether an implementation meets the specification by checking a refinement partial order on processes; we refer to this verification approach as refinement checking.
Typical verification problems for multi-component systems are PSPACE-complete (see e.g. [17]). Several structured verification approaches can be used to alleviate state explosion in refinement checking by breaking the verification of a large system into several sub-tasks, and establishing refinement individually for each subtask. One structured verification approach is hierarchical verification, where circuit descriptions are provided at several different levels of abstraction. At each level, the circuit is treated as an interconnection of modules. Refinement is checked only between successive levels: the higher level is treated as specification and the lower level is treated as implementation. The description of circuit behavior at a level between the top and bottom levels is called an intermediate specification. Hierarchical verification can reduce computational costs if the verification tasks at successive levels involve fewer components than the overall verification task. Another structured verification approach is assume-guarantee reasoning, which is essentially an induction-like argument that breaks the circularity of mutual reliance of components in a system.

The application of hierarchical verification depends on finding suitable abstractions for components, which is difficult for systems that have poor modularity. For example, peephole optimizations in digital circuits often introduce additional dependencies between the optimized submodules and the rest of the system, making it difficult to obtain suitable abstractions for the application of hierarchical verification.

Correspondingly, we propose verification techniques that adapt assume-guarantee and hierarchical verification rules to the verification of peephole optimizations. Using relative timing constraints as optimization assumptions, we can verify a circuit after peephole optimizations by: 1) verifying the corresponding circuit before optimization, while it has high modularity; 2) verifying the local replacement between circuit before optimization and circuit after optimization. If proper constraints are given, even optimizations that change the interface of a submodule can be verified quickly based on the verification of the circuit before optimization.

The framework for formal verification used in this paper is process spaces [19, 20]. Our hierarchical verification is related to the methods in [5] and [6]. Assume-guarantee rules are addressed in [5], [4], [15], [10], [11]. We extend the application of such rules by introducing optimization assumptions in verification. A distinctive point of our approach is that arbitrary processes can be used as optimization assumptions, regardless of connectivity; the choice of such processes affects the result mainly through computational costs, as explained in Section 3. In particular, our optimization assumptions can be processes representing relative timing constraints. Also, our proposed verification technique does not rely on induction properties; the proof of Theorem 1 is given without reference to the structure of executions.
2. PRELIMINARIES

2.1 Process Spaces

Process spaces [19, 20] are a general theory of concurrency, parameterized by the execution type. Systems are represented in terms of their possible executions, which can be taken to be sequences of events, functions of time, etc., depending on the level of detail desired in the analysis. In this paper, executions are taken to be traces (sequences of events); for trace executions, close relationships exist to several previous treatments of concurrency, such as [6] and [9].

Let \( E \) be the set of all possible executions. A process \( p \) is a pair \((X, Y)\) of subsets of \( E \) such that \( X \triangleleft Y = E \). A process represents a contract between a device and its environment, from the device viewpoint. Executions in \( X \triangleleft Y \), called goals, denoted by \( g_p \), are legal for both the device and the environment. Executions from outside \( X \), called escapes, denoted by \( e_p \), represent bad behavior on the part of the device. Finally, executions from outside \( Y \), called rejects, denoted by \( r_p \), represent bad behavior on the part of the environment. We also use \( a_p \) (accessible) and \( at_p \) (acceptable) to denote \( X \) and \( Y \) respectively.

![Figure 1. Example processes: (a) The C-element symbol we use; (b) Waveform; (c) Hazard-intolerant model; (d) Inertial model.](image)

Process spaces can be used to build models of circuit behavior in a manner similar to conventional state machines. For an example of the models used in this paper, consider the C-element in Fig. 1 (a). If the inputs \( a \) and \( b \) have the same logical value, the C-element copies that value at the output \( c \); otherwise, the output value remains unchanged. Waveforms are represented by finite sequences of actions corresponding to signal transitions, such as \( abcba \) for the waveform in Fig. 1 (b). In this paper, we use the term trace to refer to such a sequence of actions. We sometimes indicate that a certain action represents a rising or falling transition, as in \( a+ b+ c+ b- a- c- \).

If all signals start low, the C-element can be represented by the process in Fig. 1 (c), where \( r, g, \) and \( e \) stand for reject, goal, and escape. Illegal output events lead to an escape state with self loops on all subsequent events, call it a permanent escape, and illegal input events lead to a reject state that cannot
be left either, call it a *permanent reject*. The state where \( ab \) leads is also marked \( e \), making it illegal for the device to complete its operation by stopping there.

The model in Fig. 1 (c) is a *hazard-intolerant* model. There are variations of the CMOS cell models, because, in the presence of hazards, the behavior of a CMOS cell is not fully standardized. A hazard is a situation where an output transition is enabled and then disabled without being completed. For example, execution \( abb \) is a hazard for the C-element in Fig. 1. Hazard-intolerant models simply require the environment to avoid hazards, by stating that each execution that includes a hazard will lead to a permanent reject. The model in Fig. 1 (d) is an *inertial* model. Inertial models ignore hazards by filtering out input pulses that are shorter than the delay of the gate.

In process spaces, processes can be used to model not only gates or cells, but also relative timing assumptions of the following form:

\[
D (b_1 b_2 ... b_n) > D (a_1 a_2 ... a_m)
\]

where \( a_1, ..., a_m, b_1, ..., b_n \) are events such that \( a_1 \) is the same as \( b_1 \), and the \( Ds \) are the durations of the chains of events. Such a constraint, called a *chain constraint* [17], enforces that the \( b \) chain of events will not be completed before the \( a \) chain (unless one of the \( a \) or \( b \) actions involved occurs out of order).

Treating constraints as processes rather than linear inequalities permits us to deal with cases of deadlock and non-determinism, where the inequalities might not apply. Chain constraints can be implemented by transistor sizing. The absence of numerical information in the process models for chain constraints leads to more efficient verification using existing tools for non-timed analysis. Metric-free verification under relative timing constraints was presented in [19] and [17].

In this paper, we only use the following operations and conditions on processes:

- **Refinement** is a binary relation, written \( p \leq q \), meaning “process \( q \) is a satisfactory substitute for process \( p \)”.

- **Product** is a binary operation, written \( p \circ q \), yielding a process for a system of two devices operating “jointly”. Product is defined by \( \text{as} (p \circ q) = \text{as} p \circ \text{as} q \) and \( g (p \circ q) = g p \circ g q \).

- **Robustness** is a class of processes, written \( R_E \), so that process \( p \) is in \( R_E \) if and only if \( r p = ? \). Robustness represents a notion of absolute correctness: the device is “fool-proof” and can operate in any environment.

- **Reflection**, written \( -p \), defined by \( \text{as} (-p) = \text{at} p \) and \( \text{at} (-p) = \text{as} p \), represents a swap of roles between environment and device.
In process spaces, refinement is reflexive, transitive, and antisymmetric; product is commutative, associative, and idempotent. Furthermore, for processes \( p, q, \) and \( r \),

\[
p \preceq q \quad \Rightarrow \quad p \preceq r \preceq q \preceq r.
\]

These properties suffice to break a verification problem into several layers of partial specifications, and each layer into several modules, and to verify only one module at a time instead of the overall problem in one piece.

Product admits an identity element, which is the process \( \varepsilon \) defined by \( g \varepsilon = E \). Note that a process is robust if and only if that process refines \( \varepsilon \).

Manipulations of finite-word processes are implemented by a BDD-based tool called FIREMAPS [17] (for finitary and regular manipulation of processes and systems). FIREMAPS implements the process space operations and conditions mentioned above, and has built-in constructors for hazard intolerant and inertial models, and for chain constraints. In addition, if refinement does not hold, FIREMAPS can produce a witness execution that pinpoints the failure. Such witness executions are used for fault diagnosis.

2.2 Peephole Optimizations in Asynchronous Circuits

By peephole optimizations we mean local changes in circuit sub-modules that do not affect the rest of the circuit or the operation of the circuit as a whole. This sometimes involves changes in the interface of the respective sub-module to take advantage of signals available in other modules.

Peephole optimizations are often performed after high-level synthesis with significant gains. In [7], a peephole optimization step after synthesis by the methods in [1] and [3] is shown to produce gate count improvements up to a factor of five, and speed (cycle time) improvements up to a factor of two.

Due to their heuristic nature and limited scope, peephole optimizations can greatly benefit from automated verification. As unconventional designs, it is important that such optimizations be proven correct. Because of the small number of components involved, state explosion can be alleviated by localizing the verification.

On the other hand, flat verification of peephole optimizations against module specifications poses special problems. First, such verifications must usually take into account intricacies of switch-level and relative-time behaviors. Second, changes in the module interfaces need special techniques for modeling and incorporating the assumptions that justified the optimisations.
3. A STRUCTURED APPROACH FOR PEEPHOLE VERIFICATION

3.1 Assume-Guarantee Verification

Using hierarchical verification, a verification task $p_1 ? p_2 \preceq q$ can be split into three separate tasks that might have lower computational costs overall:

$$q_1 ? q_2 \preceq q; p_1 \preceq q_1; p_2 \preceq q_2. \quad (3-1)$$

Here, $p_1$ and $p_2$ represent two submodules in the implementation, $q_1$ and $q_2$ represent the intermediate specifications of the respective submodules, and $q$ represents the overall specification implemented by $p_1$ and $p_2$.

A frequently encountered difficulty, however, is loss of context information if operating assumptions are not modeled. Modules $p_1$ and $p_2$ may only “work correctly” in the presence of each other, in which case the refinement relations $p_1 \preceq q_1$ and $p_2 \preceq q_2$ only hold under certain assumptions. In Fig. 2, the OR gate does not directly refine the XOR gate specification. Nevertheless, if the environment guarantees that the inputs are mutually exclusive, a refinement check in context holds, verifying that the OR gate safely implements the XOR; see Example 1 in Subsection 3.2.

![Figure 2. Replacement under assumptions: (a) XOR in a certain environment; (b) Inertial XOR; (c) OR in a certain environment.](image)

In verification methods based on assume-guarantee rules (such as [5], [15], [10], and [11]), the correctness of the implementation relies on assumptions from the environment. According to assume-guarantee rules, a verification of the form $p_1 ? p_2 \preceq q$ can be decomposed as follows:

$$p_1 ? q_2 \preceq q ? q_1; \quad (3-2)$$
$$p_2 ? q_1 \preceq q ? q_2. \quad (3-3)$$

More precisely, it suffices to verify (3-2) and (3-3) for some $q_1$ and $q_2$ of a certain restricted form in order to establish $p_1 ? p_2 \preceq q$. However, note that $q_1$ and $q_2$ cannot be arbitrary. For instance, empty accessible sets of $q_1$ and $q_2$ trivially satisfy the decomposed verification tasks. Special conditions must be satisfied by $q_1$ and $q_2$ to break the circularity of reasoning and establish
validity of \( p_1 \preceq p_2 \preceq q \). In previous methods, \( q_1 \) and \( q_2 \) are selected to have non-empty prefix-closed accessible sets, which justify a structural induction argument. In our method, we extend the rule so that \( q_1 \) and \( q_2 \) are arbitrary.

### 3.2 The Peephole Rule

Suppose that, in the original verification task, the refinement relation

\[
p_1 ? \ldots r_i ? \ldots ? p_n \preceq q
\]

is checked. Here \( p_i (i=1..n) \) and \( q \) are arbitrary processes, representing the implementation components and specification, respectively; \( r_i \) is an arbitrary process, call it peephole replacement, which replaces \( p_i \) in a peephole optimization. Now, let \( d \) be an arbitrary process, call it optimization assumption, which formalizes the designer’s hypothesis that made the replacement possible; and, let \( M \) be an arbitrary set of processes, call it support model, representing modules in the closed system, consisting of the implementation and the environment, that will guarantee the validity of the optimization assumption.

**Theorem 1.** For processes \( p_1, \ldots, p_n, q, r_i, d \) and process set \( M \):

\[
\text{if } p_1 ? \ldots r_i ? \ldots ? p_n \preceq q \\
\text{and } (? d? S_E, M ? \{p_j \mid j \neq i\} \preceq \{\neg q\}:
\text{ } r_i ? d \preceq p_i ? (?_{m \neq M} M) \preceq (?_{m \neq M} M) ? d)
\text{ then } p_1 ? \ldots r_i ? \ldots ? p_n \preceq q.
\]

We can phrase Theorem 1 informally as follows: if \( r_i \) refines \( p_i \) under constraint \( d \), and \( d \) imposes no additional confinement over the system, then \( r_i \) can replace \( p_i \) in a refinement relation.

**Proof:**

\[
\begin{align*}
\text{r}_i ? d & \preceq \text{p}_i \\
\text{? ? (by monotonicity w.r.t. product by } (\text{?}_{j \neq i} \text{p}_j) ? (\neg q)) \\
(\text{?}_{j \neq i} \text{p}_j) ? (\neg q) & ? r_i ? d \preceq (\text{?}_{j \neq i} \text{p}_j) ? (\neg q) ? \text{p}_i \\
\text{? ? (by commutativity of product)} \\
(\text{?}_{j \neq i} \text{p}_j) ? (\neg q) & ? r_i ? d \preceq (\text{?}_k \text{p}_k) ? (\neg q)
\end{align*}
\]

(1)

\[
\begin{align*}
(\text{?}_{m \neq M} M) & \preceq (\text{?}_{m \neq M} M) ? d \text{ and } M ? \{p_j \mid j \neq i\} \preceq \{\neg q\} \\
\text{? ? (by monotonicity of product w.r.t. components from outside } M) \\
(\text{?}_{j \neq i} \text{p}_j) ? (\neg q) & \preceq (\text{?}_{j \neq i} \text{p}_j) ? (\neg q) ? d \\
\text{? ? (by monotonicity of product w.r.t. } r_i) \\
(\text{?}_{j \neq i} \text{p}_j) ? (\neg q) & ? r_i \preceq (\text{?}_{j \neq i} \text{p}_j) ? (\neg q) ? d ? r_i
\end{align*}
\]

(2)
Since Theorem 1 is specifically developed for verifying peephole optimizations, we refer to Theorem 1 as the peephole rule.

Example 1: The example in Fig. 2 illustrates a peephole optimization that replaces an XOR gate by an OR gate in a mutual exclusion environment. In this example, the peephole implementation is an inertial XOR gate; the peephole replacement is an inertial OR gate; the optimization assumption is “Pulses on $a$ and $b$ never overlap”; and, the support model is a Mutex component existing in the system.

In the verification of Fig. 2, assume refinement relation

$$p_{\text{XOR}} \ldots p_{\text{mutex}} \ldots \leq q$$

holds, in which $p_{\text{XOR}}$ represents the XOR gate and $p_{\text{mutex}}$ represents an environment component which physically enforces mutual exclusion of signals $a$ and $b$.

Let $M = \{p_{\text{mutex}}\}$, and let $d$ be mutual exclusion constraint in Fig. 3(b). Note that $d$ is the product of the following relative timing constraints:

$$D(a+ a-) < D(a + b+);$$
$$D(b+ b-) < D(b+ a+).$$
Because
\[ p_{\text{mutex}} \preceq p_{\text{mutex}} ? d \]
and
\[ p_{\text{OR}} ? d \preceq p_{\text{XOR}}, \]
using the peephole rule, we have:
\[ p_{\text{OR}} ? \cdots p_{\text{mutex}} ? \cdots \preceq q. \]
Thus, the OR gate can safely replace the XOR gate in this system.

Theorem 1 relates to the assume-guarantee rule over a particular case. If \( p_1 \) from (3-2) is substituted by \( r_i \) from Theorem 1, \( q_1 \) from (3-2) is substituted by \( p_i \) from Theorem 1, and \( q_2 \) from (3-2) is substituted by the process \((?_{j=1} p_j)\), then the hypothesis of Theorem 1 implies inequation (3-2), which is part of the hypothesis of the assume-guarantee rule.

The remaining part of the hypothesis of the assume-guarantee rule, inequation (3-3), also follows from the hypothesis of Theorem 1 under the substitutions above, while \( p_2 \) from (3-2) can be substituted by any process \( p \) that refines \((?_{j=1} p_j)\).

Theorem 1 also relates to hierarchical verification over a particular case. If \( p_j \) is the intermediate specification for \( r_i \), and \( M \) is the empty set, then \((?_{m \in M} m) = ?\). Since \( ? \) is the most transparent process in a process space (Definition 2.13 in [17]) then \( d = ? \). In this case, repeated application of Theorem 1 matches the hierarchical verification procedure as described by inequation (3-1).

### 3.3 Heuristics for Finding Verification Assumptions

Not all the optimization assumptions are guaranteed by the environment of the peephole alone. Since Theorem 1 has no restrictions on the connectivity of the processes involved, an optimization assumption may overlap both the peephole and the peephole environment. If the optimization assumption overlaps the peephole, we call the respective optimization assumption a design assumption; otherwise, we call the respective optimization assumption an environment assumption. Design assumptions are derived from properties of the circuit under verification which are known to the designer or verifier.

Notice that the peephole rule has no restrictions on the choice of the support model or the optimization constraint: system process subset \( M \) and process \( d \) in Theorem 1 are arbitrary. On the other hand, a poor selection of the support model will not lead to reductions of computational costs. For
proper selection of the support model, one should consider the context of the circuit under verification. For example, in Fig. 3(c), the process of “OR gate under environment assumption” has fewer accessible executions than the process in Fig. 3(a) “OR gate in open system”, because some of the possible behaviors are eliminated by an optimization assumption based on a support model which involves not just circuit elements, but also the environment of the circuit.

In our experiments, the costs of verification based on peephole rules are influenced mainly by a tradeoff between the complexity and the determinism of the assumptions used.

?? Keep the assumptions simple, as the complexity of verification increases with the number of assumption processes.

?? Make the assumptions efficient by eliminating as many as possible of the “don’t-care behaviors” of the circuit under verification.

For instance, in the OR gate example above, a more effective assumption would only allow the pulses of \( a \) and \( b \) to alternate with pulses on \( c \). If such an assumption can be guaranteed by the environment of the peephole, then not only we can perform stronger optimizations, but also we can verify them with less costs.

Presently, we mostly use relative timing constraints as optimization assumptions. By using relative timing constraints, verification does not need to start from a complete environment model, because a few hints from an incomplete environment model may suffice to guarantee the respective delay constraint as optimization assumption.

4. VERIFICATION OF THE ASP* ARBITER

A high-speed arbiter using the asP* protocol is reported in [8], with a non-optimized version and a speed-optimized version. The non-optimized implementation has good modularity, in the sense that the simple interfaces of the submodules achieve decoupling of the submodule designs. The optimized implementation achieves higher speed at some costs in modularity, by including more signals in the submodule interfaces. The high-level verification of the non-optimized version of this arbiter was reported in [13]. In this paper, we focus on the verification of the peephole optimisations.

4.1 Verification Strategy for the asP* Arbiter

The block diagram of the non-optimized version of the arbiter from [8] is shown in Fig. 4(a). The arbiter receives request events as input pulses and
issues grant events as output pulses, after arbitration. Signals $q_1$ and $q_2$ are initially high, and the other signals are initially low. The rlatch component is a positive edge-triggered SR latch. When it receives input pulses ($r_1$ or $g_1$ for rlatch1), the rlatch converts them to output signal levels ($y_1$ for rlatch1). The dlatch in Fig. 4(a) converts pulses on the inputs to levels on the output. The two NOR gates generate grant pulses. The Mutex component ensures mutual exclusion between requests, and it is a four-phase component. For example, when request pulses from two channels arrive, rlatch1 and rlatch2 set $y_1$ and $y_2$ high. The Mutex arbitrates the input and gives grant to one, e.g. channel1, then $q_1$ is set to low and fires the rising edge of $g_1$. The rising edge of $g_1$ is propagated to dlatch and set $f$ high, thereby resetting $g_1$; at the same time, feedback of $g_1$ pulse withdraws the request $y_1$. The pending request from channel2 gets a grant from Mutex and grant pulse $g_2$ to channel2 will be issued after an acknowledge pulse $d_1$ from channel1 is received. Notice that the grant pulse will be issued without waiting for the falling edge on the request pulse; only the rising edges of the $r$, $g$, and $d$ signals are active.

To prove correctness of this arbiter, we aim to establish refinement between the high-level specification and the switch-level implementation. We apply the hierarchical verification procedure described in Section 3, using the submodules in Fig. 4(a) as intermediate specifications. In the following, $p_{Mutex}$, $p_{rlatch1}$, $p_{rlatch2}$, $p_{dlatch}$, $p_{NOR1}$ and $p_{NOR2}$ denote processes for the switch-level implementation of submodules, $q_{Mutex}$, $q_{rlatch1}$, $q_{rlatch2}$, $q_{dlatch}$, $q_{NOR1}$ and $q_{NOR2}$ denote processes for the submodule interfaces, and $q_{arbiter}$ denotes the process for the overall transition-event specification of the arbiter. The verification proceeds on two levels, as follows:

On the first level, the refinement between the connection of intermediate specifications and overall specification is checked:

$q_{Mutex} \Rightarrow q_{rlatch1} \Rightarrow q_{rlatch2} \Rightarrow q_{dlatch} \Rightarrow q_{NOR1} \Rightarrow q_{NOR2} \ll q_{arbiter}$
Next, the refinements of submodule specifications against their switch level implementations are checked under certain environment assumptions:

\[
\begin{align*}
    p_{\text{Mutex}} & \leq q_{\text{Mutex}}; \\
    p_{\text{rlatch}} & \leq q_{\text{rlatch}}; \\
    p_{\text{dlatch}} & \leq q_{\text{dlatch}}; \\
    p_{\text{NOR}} & \leq q_{\text{NOR}}.
\end{align*}
\]

In this paper, we call the verification on the first level high-level verification and the second level submodule verification. For the details of high-level verification and overall specification, we refer readers to [13].

### 4.2 Rlatch Verification

Submodule verification consists of refinement checks between switch level implementation of submodules and their intermediate specifications. We use rlatch as the example of refinement checking on this level. The representations of implementations at switch level follow [17].

The implementation of the rlatch from [8] is shown in the Fig. 4 (b). When a rising edge of signal \( \text{set} \) comes, the inverter chain of set (contains 4 inverters) generates a pulse upon the PMOS network, thus setting output \( y \) high. The same mechanism is used for reset. The box labeled \( K \) is a “keeper” circuit consisting of a weak inverter and a feedback inverter that form a loop, so that the value of the rlatch output can be kept when there are no set or reset paths enabled.

The rlatch is verified by checking the following refinement relationships:

\[
p_{\text{rlatch}} = p_{\text{set inv chain}} \circ p_{\text{reset inv chain}} \circ q_{\text{MOS}} \leq q_{\text{rlatch}}
\]

in which \( p_{\text{set inv chain}} \) is the model of set inverter chain (4 inverters); \( p_{\text{reset inv chain}} \) is the model of reset inverter chain (3 inverters); \( p_{\text{MOS}} \) is the model of the reset part of rlatch.

Some assumptions can be drawn from the environment of the rlatch in the asP* arbiter system. Interface signals \( \text{set} \), \( \text{reset} \) and \( y \) of the rlatch are under constraints. For instance, in the channel1 of asP* arbiter, grant signal \( g_1 \) is used as reset signal of rlatch1. Notice that for rlatch1, \( g_1 + \) will not be triggered until \( y_1 + \) is propagated through Mutex and NOR gate. As a result, we can make an environment assumption for rlatch:

\[
d_{\text{env}}: D (\text{set}+ \text{reset}+) > D (\text{set}+ y+)
\]

Correspondingly, the verification of the rlatch becomes:

\[
d_{\text{env}} \circ p_{\text{rlatch}} \leq q_{\text{rlatch}}
\]

The verification result shows that the refinement relationship holds when the input pulse is wide enough to be caught by the inverter chain. This relative timing constraint is easily implemented by sizing, thus the rlatch
implementation in Fig. 4(b) refines rlatch specification in typical environments for the asP* arbiter.

### 4.3 Peephole Optimizations of the asP* Arbiter

Fig. 5 (a) is the block diagram of asP* arbiter after optimization in [8]. Notice the change of submodules at their interfaces. (For example, the new interface of Mutex has 12 signals.) In Fig. 5 (a), signal $rr$ of the Half_Mutex (Mutex/2 blocks in Fig. 5(a)) is connected to the internal nodes $set_2$ of rlatch (see fig. 4(b)); signal $gb$ is connected to the internal node $set_1$ of dlatch, which has the same setting-resetting structure as that for the rlatch [8].

**Figure 5.** Peephole Optimization of asP* arbiter: (a) asP* arbiter block diagram after optimization; (b) Half-Mutex after optimisation.

Fig. 5 (b) shows the peephole optimizations over the half Mutex. Boxes in dark indicate modifications during the optimizations. Signals $rr$ and $gb$ from the internal nodes of rlatch and dlatch are highlighted. For the details and motivation of these optimizations, we refer the reader to [8].

Notice that only Mutex’s low-level structure is changed by optimization. For the other submodules, optimization only changes the interfaces by making some internal signals visible. As shown in Fig. 6(a), modules $r1$, $rr2$, $gb1$ and $gb2$ are “abstracted” from rlatch1, rlatch2 and dlatch. Relations between models hold as follows:

$$p_{rlatch1} = p_{r1} \land p_{rlatch1} ; \ p_{rlatch2} = p_{rr2} \land p_{rlatch2} ; \ p_{dlatch} = p_{gb1} \land p_{gb2} \land p_{dlatch}$$

By this module partition, we isolate unchanged modules (white blocks) in Fig. 4(a) from the models changed by optimization (dark blocks). The verification task after peephole optimization verification is:

$$p_{gb1} \land p_{gb2} \land p_{r1} \land p_{rr2} \land p_{ Mutex' } \land q_{rlatch1} \land q_{rlatch2} \land q_{dlatch} \land q_{ NOR1 } \land q_{ NOR2 } \leq q_{arbiter}$$

In which $p_{ Mutex' }$ represents the replacement of $p_{ Mutex }$ in optimization.
4.4 Verification of Peephole Optimizations

Applying the peephole rule, the verification of the peephole optimised Mutex amounts to verifying the following relation, for some constraint \( d \):

\[
P_{Mutex} \cdot d \leq q_{Mutex}
\]

and to verifying that constraint \( d \) does not impose undue confinement on the system, by finding a suitable support model for \( d \).

There are three optimizations addressed in [8]. The first optimization over half-Mutex is shown in Fig. 6(b). Changes involved in this optimization are highlighted. To apply the peephole rule, we take the following preparation steps:

?? First, a model of the optimization assumption is constructed. In Fig. 6(b), the highlighted part indicates the optimization assumption of optimization, which "adds an additional 'bypass' to allow the rising edge of a request to be applied directly to the arbiter without incurring the delay of the latch" [8]. The relative timing constraint \( d_1: D(r_i+nq_i-) > D(r_i+y_i+)(i = 1, 2) \) in Fig. 6(b) comes from this statement.

?? Second, the refinement

\[
p_{gb1} \cdot p_{gb2} \cdot p_{rr1} \cdot p_{rr2} \cdot q_{Mutex} \cdot q_{rlatch1} \cdot q_{rlatch2} \cdot q_{rlatch} \cdot q_{NOR1}
\]

?? \( q_{NOR2} \cdot d_1 \leq q_{arbiter} \)

is checked and the relation holds.

?? In the last step, we choose \( M = \{p_{gb1}, p_{gb2}, p_{rr1}, p_{rr2}, d_1\} \) as the initial support model of peephole verification, and optimization assumption \( d = d_1 \cdot p_{rr1} \cdot p_{rr2} \).
Two constraints are introduced in the procedure. One constraint:

\[ d_2: \ D(r_i^+ g_r^+) > D(r_i^+ y_i^+) \ (i = 1, 2) \]

has as support model the components \( p_{rlatch_1} \) and \( p_{rlatch_2} \). The result of peephole verification is:

\[
M = \{ p_{gb1}, p_{gb2}, p_{rr1}, p_{rr2}, d_1, q_{rlatch1}, q_{rlatch2} \}; \\
d = d_1^? d_{rr1}^? d_{rr2}^? d_2^? \\
p_{Mutex}^? d^? d_3 \leq q_{Mutex}
\]

in which \( d_3 \) is an extra delay constraint:

\[ d_3: \ D(nq_i^+ q_i^+) > D(nq_i^+ q_i^+) \ (i = 1, 2) \]

needed for the holding of the refinement.

The second and third optimizations are verified in the same way and the refinement only holds under certain additional constraints.

To examine the validity of the additional constraints detected in the peephole verification, we changed peephole in verification from Mutex to the whole arbiter. Applying peephole rules, we verify the implementation in Fig. 6 (a). The verification terminated and the result reported that no other relative timing constraints are needed for arbiter optimization:

\[
M = \{ d_{dest1}, d_{dest2} \}; \\
d = d_{dest1}^? d_{dest2}^? \\
p_{Mutex}^? q_{rlatch1}^? q_{rlatch2}^? q_{drlatch}^? q_{Nor1}^? q_{Nor2}^? d_b^? d \leq q_{arbiter} \\
d_{dest1}: \ D(r_i^+ nq_i^+) < D(r_i^+ y_i^+) \ \text{from optimization 1} \ (i = 1, 2) \\
d_{dest2}: \ D(g_i^+ gb_i^+) > D(g_i^+ y_i^+) \ \text{from optimization 2} \ (i = 1, 2)
\]

We conclude that the optimized arbiter in Fig. 5 (a) can substitute the arbiter in Fig. 4(a).

**Acknowledgments**

We thank Mark Greenstreet for numerous comments, and Mark Josephs for discussions that inspired Example 1. We gratefully acknowledge financial support from NSERC, FCAR, CFI, MEQ, and McGill University.

**REFERENCES**


SYMBOLIC VERIFICATION OF COMPLEX REAL-TIME SYSTEMS WITH CLOCK-RESTRICTION DIAGRAM

Farn Wang
Institute of Information Science, Academia Sinica
Taipei, Taiwan 115, ROC
farn@iis.sinica.edu.tw

Abstract
Real-world real-time systems may involve many complex structures, which are difficult to verify. We experiment with the model-checking of an application-layer html-based web-camera which involves structures like event queues, high-layer communication channels, and time-outs. To contain the complexity, we implement our verification tool with a newly developed BDD-like data-structure, reduced CRD (Clock-Restriction Diagram), which has enhanced the verification performance through intensive data-sharing in a previous report. However, the representation of reduced CRD does not allow for quick test of zone containment. To this purpose, we thus have designed a new CRD-based representation, cascade CRD, which has given us enough performance enhancement to successfully verifying several implementations of the web-camera.

Keywords: Real-time systems, BDD, verification, Clock-Restriction Diagram

1. INTRODUCTION

Fully symbolic model-checking with BDD data-structure[8, 4] has achieved great success in hardware systems and become one of the standard technologies in Electronic Design Automation (EDA). Naturally, people are now looking forward to repeating the success in model-checking complex systems, which may involve structures like queues, stacks, clocks, communicaiton channels, . . . . Such structures usually cause great complexities to verify with BDD-like data-structures. In this work, we experiment with a real-world project, an html-based web-camera (http://www.mmedia.com.tw) implemented in application layer, and discuss how we have coped with the challenges in verification. Especially, we implemented the newly developed CRD (Clock-Restriction Diagram), which is a BDD-like data-structure for space-efficiency in timed automaton verification, in our model-checking tool and found that the representation
of reduced CRD does not allow for efficient testing of zone containment. It was only with our new representation scheme of cascade CRD that we had been able to verify implementations of the web-cameras.

Most modern model-checkers for real-time systems are built around some symbolic manipulation procedures\cite{12} of zones implemented in data-structures like DBM\cite{10}, NDD\cite{1}, CDD\cite{7}, RED\cite{15}, or CRD\cite{16}. A zone means a behaviorally equivalent state subspace of a timed automaton and is symbolically represented by a set of difference constraints between clock pairs. It has been the general opinion that DBM (difference-bounded matrix) \cite{10} is perhaps the best data-structure for the representation and manipulation of zones. For a long time, BDD-like structures\cite{1, 7, 15, 16} have not performed as well as the popular DBM (difference-bounded matrix) \cite{10} which is a 2-dimensional matrix and nothing BDD-like. But recently, there comes a new BDD-like data-structure, called CRD (Clock-Restriction Diagram) \cite{17}, which has shown efficiency in experiments (see appendix 6). CRD uses evaluation variables corresponding to the entries in DBM. Particularly, the default value of variables, say \(x - x'\), is treated as \(x - x' < \infty\) (i.e. no restriction). Since a CRD is actually a DBM set represented in decision diagram, it is neither a canonical representation of dense-domain state-spaces. The representation of CRD with zones with smallest set of inequalities\cite{14}, i.e. reduced CRD, was proposed in \cite{17} to cope with space-complexity. But the drawback of reduced CRD is that zone containment cannot be determined efficiently. Thus many many zones, which are contained by others in the reachable state space representation, are calculated again and again. As a result, both space and time-complexities become unmanageable and our verification tool with reduced CRD could not finish verifying several implementations of the web-camera.

Note that the problem, with the zone containment relation, mentioned in the last paragraph cannot be simply handled by using the representation of CRD with all-pair shortest-path representation zones \cite{10, 14} because as argued in \cite{17}, such representation is less space-efficient than reduced CRD. To overcome this obstacle, we here propose a new representation of CRD, the cascade CRD, which is not a representation of set of zones with minimal number of inequalities. For a zone with identical clock readings, cascade CRD may contain more inequalities than reduced CRD. But for a given state-space (i.e. a set of zones), cascade CRD may result in much less zones recorded than reduced CRD with efficient elimination of zones contained by others. It is with the cascade CRD that we have successfully verified several implementations of the web-camera system.

Here is our presentation plan. Section 2 briefly defines timed automata as our model for discussion. Section 3 basically restates part of \cite{17} to give readers a background knowledge to the issues in verification research of timed automata and CRD. Section 4 presents cascade CRD and its computation. Section 5
reports our tool implementations and experiments with the web-camera. Appendix 6 shows how good our CRD-based verification tool is compared to two well-known verification tools and justifies our choice to continue experiments with CRD.

2. TIMED AUTOMATA

We use the widely accepted timed automata[2] as our model. A timed automaton is a finite-state automaton equipped with a finite set of clocks which can hold nonnegative real-values. It is structured as a directed graph whose nodes are modes (control locations) and whose arcs are transitions. The modes are labeled with invariance conditions while the transitions are labeled with triggering conditions and a set of clocks to be reset during the transitions. The invariance conditions and triggering conditions are Boolean combinations of inequalities comparing a clock with an integer. At any moment, the timed automaton can stay in only one mode. In its operation, one of the transitions can be triggered when the corresponding triggering condition is satisfied. Upon being triggered, the automaton instantaneously transits from one mode to another and resets clocks in the corresponding transition clock set label to zero. In between transitions, all clocks increase their readings at a uniform rate.

For convenience, given a set $X$ of clocks, we use $B(X)$ as the set of all Boolean combinations of inequalities of the form $x - x' \sim c$ where $x, x' \in X \cup \{0\}$, “$\sim$” is one of $\leq, <, =, >, \geq$, and $c$ is an integer constant.

Definition 1 automata A timed automaton $A$ is given as a tuple $\langle X, Q, q_0, I, \mu, T, \gamma, \tau, \pi \rangle$ with the following restrictions. $X$ is the set of clocks. $Q$ is the set of modes. $q_0$ is the initial mode. $I \in B(X)$ is the initial condition on clocks. $\mu: Q \rightarrow B(X)$ defines the invariance condition of each mode. $T$ is a finite set of transitions. $\gamma: T \rightarrow (Q \times Q)$ describes the source and destination modes of transitions. $\tau: T \rightarrow B(X)$ and $\pi: T \rightarrow 2^X$ respectively defines the triggering condition and the clock set to reset of each transition.

A valuation of a set is a mapping from the set to another set. Given an $\eta \in B(X)$ and a valuation $\nu$ of $X$, we say $\nu$ satisfies $\eta$, in symbols $\nu \models \eta$, iff when the variables in $\eta$ is interpreted according to $\nu$, $\eta$ will be evaluated true.

Definition 2 states Given a timed automaton $A = \langle X, Q, q_0, I, \mu, T, \gamma, \tau, \pi \rangle$, A state $\nu$ of $A$ is a valuation of $X \cup \{\text{mode}\}$ such that

- $\nu(\text{mode}) \in Q$ is the mode of $A$ in $\nu$ with mode as a special auxiliary variable; and
- for each $x \in X$, $\nu(x) \in R^+$ such that $R^+$ is the set of nonnegative real numbers and $\nu \models \mu(\nu(\text{mode}))$. 

||
For any \( t \in R^+ \), \( v + t \) is a state identical to \( v \) except that for every clock \( x \in X \),
\[ v(x) + t = (v + t)(x). \] Given \( \vec{X} \subseteq X \), \( v \vec{X} \) is a new state identical to \( v \) except
that for every \( x \in \vec{X} \), \( v \vec{X}(x) = 0 \).

**Definition 3 runs**

Given a timed automaton \( A = \langle X, Q, q_0, I, \mu, T, \gamma, \tau, \pi \rangle \), a \( v \)-run is an infinite sequence of state-time pair \((v_0, t_0) - (v_1, t_1) - \ldots \ldots \)

such that \( v = v_0 \) and \( t_0 = 0 \) . . . . . . \( v_k \) . . . . is a monotonically increasing real-
number (time) divergent sequence, and for all \( k \geq 0 \),

- for all \( t \in [0, t_{k+1} - t_k] \), \( v_k + t \models \mu(v_k (\text{mode})) \); and
- either \( v_k (\text{mode}) = v_{k+1} (\text{mode}) \) and \( v_k + (t_{k+1} - t_k) = v_{k+1} \); or for

some \( w \in T \),
  \[ - \gamma(w) = (v_k (\text{mode}), v_{k+1} (\text{mode})); \]
  \[ - v_k + (t_{k+1} - t_k) \models \tau(w); \] and
  \[ - (v_k + (t_{k+1} - t_k)) \pi(w) = v_{k+1}. \] 

A safety requirement on timed automaton \( A \) can be written as a Boolean
combination of clock constraints in \( B(X) \) and mode restrictions in the form
of \( \text{mode} = q \) meaning that \( A \) is currently in mode \( q \in Q \). A run \( \rho = (v_0, t_0)(v_1, t_1) \ldots \ldots \)
of \( A \) satisfies safety requirement \( \eta \), in symbols \( \rho \models \eta \), iff for all \( k \geq 0 \) and \( t_k \leq t \leq t_{k+1} \), \( v_k + t \models \eta \). We say \( A \models \eta \)
iff for all \( v \)-runs \( \rho, v \models I \wedge (\text{mode} = q_0) \) implies \( \rho \models \eta \). Our verification
framework is safety analysis problem that when given \( A \) and \( \eta \), asks whether
\( A \models \eta \).

### 3. BASICS

To better prepare readers for understanding of the materials, we restate some
of the paragraphs form [17] in this section. Subsection 3.1 discusses the concept
of zones. Subsection 3.2 formally defines CRD and its manipulations. Espe-
\[ \begin{align*}
\text{Let } \mathcal{Z} \text{ be the set of integers. Given } c \geq 0 \text{ and } c \in \mathcal{Z}, \text{ let } I_c \text{ be } \{ \infty \} \cup \{ d \mid d \in \mathcal{Z}, -c \leq d \leq c \}. \text{ Also for any } d \in \mathcal{Z}, d + \infty = \infty + d = \infty.
\end{align*} \]
Given a safety analysis problem for a timed automaton $A$ with biggest timing constant $C_A$ used in $A$, a zone is a convex subspace of $R^{\|X\|}$ constrained by half spaces represented by inequalities like $x - x' \sim d$, with $x, x' \in X \cup \{0\}$, $\sim \in \{\leq, <\}$, and $d \in I_{C_A}$, such that when $d = \infty$, $\sim$ must be "$<\$". For convenience, let $B_c = \{(\sim, d) \mid \sim \in \{\leq, <\}, d \in I_c, d = \infty \Rightarrow \sim = "<"\}.

With respect to given $X$ and $C_A$, the set of all zones is finite. Formally, a zone $\zeta$ can be defined as a mapping $(X \cup \{0\})^2 \mapsto B_{C_A}$. Alternatively, we may also define a zone $\zeta$ as the set $\{x - x' \sim d \mid \zeta(x, x') = (\sim, d)\}$. In the following, we shall use the two equivalent definitions flexibly as we see fit.

There can be many zones representing the same convex subspace. A straightforward canonical representation of a zone-characterizable convex subspace is its zone in closure form (called shortest-path closure in [14]). A zone $\zeta$ is in closure form if and only if for any sequence of elements $x_1, \ldots, x_k \in X \cup \{0\}$, with $x_1 - x_k \sim d \in \zeta$ and $\forall 1 \leq i < k (x_i - x_{i+1} \sim_i d_i \in \zeta)$, either $d < \sum_{1 \leq i < k} d_i$ or $(d = \sum_{1 \leq i < k} d_i \land (\sim = \"\leq\" \Rightarrow \land_{1 \leq i < k} \sim_i = \"\leq\"))$. Intuitively, this means that every half space constraint has to be tight. We can artificially designate the closure form of each zone as our canonical representation of the corresponding state subspace characterized by the zone. For convenience, given a zone $\zeta$, we let $\zeta^C$ be the notation for its closure form.

A few terms to define before we explain the second candidate for zone canonical representation. Two clocks $x, x' \in X \cup \{0\}$ are equivalent in a zone $\zeta$, in symbols $x \equiv_\zeta x'$, iff $\exists d \in Z(x - x' \leq -d \in \zeta^C \land x' - x \leq d \in \zeta^C)$. For convenience, assume that $X = \{x_1, \ldots, x_n\}$ and 0 is also named $x_0$. If $Y = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \subseteq X \cup \{0\}$ is a maximal set of equivalent clocks in $\zeta$ such that $i_1 < i_2 < \ldots < i_k$, let $\min_{\equiv_\zeta}(x_{i_j}) = x_{i_1}$ for all $1 \leq j \leq k$. Given $x - x' \sim d \in \zeta^C$, $(x, x')$ is redundant in $\zeta$ iff $x \not\equiv_\zeta x'$ and there is a $\bar{x} \in X \cup \{0\}$, with $x - \bar{x} \sim_\zeta d_1$, $\bar{x} - x' \sim_\zeta d_2 \in \zeta^C$, such that $x \not\equiv_\zeta \bar{x} \not\equiv_\zeta x'$ and $d = d_1 + d_2 \land (\sim_\zeta = "<" \Rightarrow (\sim_\zeta = "<" \lor \sim_\zeta = "\leq"))$.

Another candidate for the canonical representation of zones is the reduced form (called shortest-path reduction in [14]) which records only minimum number of constraints for each zone. We refer interested readers to [14, 17] for explanation how to convert a given zone $V$ to its zone in reduced form, in symbols $\zeta^R$. It is shown in [14] that $\zeta^C = (\zeta^R)^C$; and DBM with zones in reduced form can be used as a canonical representation of timed automaton convex states-spaces and can significantly save space in model-checking.

### 3.2 Clock Restriction Diagram

CRD is not a decision diagram for state space membership. Instead it is like a database for zones. We devise the new data-structure CRD exactly because CRD acts like a database (recording device) and is more suitable for comparison and manipulation of sets of clock difference constraints.
3.2.1 Previous data-structures. NDD[1] uses binary encoding for clock readings and its performance is very sensitive to timing-constant magnitude.

CDD[7] is a decision diagram for state-space membership. It has very similar structure to CRD. There are two major differences of CRD. First, a value \( d \) for evaluation variable, say \( x - x' \), in CDD means \( x - x' = d \) while a value like \( (-\infty, d) \) means \( x - x' \sim d \). This is because CDD is designed to be decision diagram. Second, the default value of a variable in CRD is interpreted as \( (-\infty, \infty) \) while it is interpreted as \( (-\infty, 0) \) in CDD. To reduce information duplication in representations of state-spaces, CDD has to be transformed to closure form [10, 7] which records the shortest-path distances between all pairs of clocks and is very space-inefficient.

RED[15, 16] encodes the ordering of fractional parts of clock readings in the variable ordering and has achieved very high space-efficiency for systems with large number of clocks and small timing constants. RED is indeed a canonical representation of timed automaton state subspaces. But for large timing constants, RED’s performance degrades rapidly.

At this moment, DBM is still the most popular and efficient data-structure. DBM-technology generally handles the complexity of timing constant magnitude very well. But when the number of clocks increases, its performance also degrades rapidly.

3.2.2 Definition. CRD is a directed acyclic graph for representation of sets of zones. It has similar structure as BDD without FALSE terminal. Each of the pairs \( (x, x') \in (X \cup \{0\})^2 \) is treated as an evaluation variable. By fixing an evaluation order, we can construct a CRD just as BDD, CDD, or RED. For example, given \( C_A = 10 \), the CRD for a set \( \{0 - x_1 \leq -3, x_1 - x_3 < -4\}, (0 - x_2 < -1, x_2 - x_1 < 6\} \) of two zones (constraints of the form \( x - x' < \infty \) are omitted) is in figure 1(a). In CRD, a missing constraints on differences of clock pairs, say \( x, x' \), is interpreted as \( x - x' < \infty \). Thus in the root vertex, even no constraint is on \( 0 - x_1 \) in the latter zone, we still construct an arc with \( 0 - x_1 < \infty \) from the root vertex. This is one major difference of our CRD from decision diagrams like CDD which interprets a missing restriction on \( x, x' \) as \( -\infty < x - x' < \infty \) with an implied lowerbound of \( -\infty \) on \( x - x' \).

An evaluation index \( \omega : (X \cup \{0\})^2 \cup \{\text{true}\} \mapsto \{0, 1, \ldots, |(X \cup \{0\})^2|\} \) is a mapping such that \( \omega(\text{true}) = \{(X \cup \{0\})^2\} \) and for every two \( e, e' \in (X \cup \{0\})^2 \cup \{\text{true}\} \), \( \omega(e) \neq \omega(e') \).

**Definition 4** Clock Restriction Diagram (CRD) A CRD is a labeled directed acyclic graph \( D = (V, \phi, E, \lambda) \), with single source and single sink, constructed under a given evaluation index \( w \) such that

- \( V \) is the set of vertices;
- \( \phi : V \mapsto (X \cup \{0\})^2 \cup \{\text{true}\} \) defines the evaluation variable at each vertex;
There is at most one $v \in V$ such that $\phi(v) = \text{true}$ and this $v$ is the single sink of the CRD.

$E \subseteq V \times V$ is the set of arcs such that for every $(v, v') \in E$, $\omega(\phi(v)) < \omega(\phi(v'))$ (i.e., evaluation ordering must be respected); and

- $\lambda : E \mapsto \mathcal{I}_{CA}$ such that for every $(v, v'), (v, v'') \in E$, $v' \neq v'' \Rightarrow \lambda(v, v') \neq \lambda(v, v'')$

There is at most one $v \in V$ such that $\phi(v) = \text{true}$ and this $v$ is the single sink of the CRD.

Since many zones can represent the same subspace, like DBM and CDD, neither is CRD a canonical representation of zone-characterized state spaces. To reduce information duplication in state-space representation, one solution is to convert all zones to their closure form \cite{10,7} (called shortest-path closure in \cite{14}), which is the set of all pairwise clock difference constraints derived from the all-pair shortest-path distances, and only store their closure form. Such conversion is expensive and, as we shall illustrate in figure 1(b)-(d), incurs large space consumption with data-structures like CDD and CRD.

Also note that although zones in closure form can be used as canonical representation for convex state-spaces characterizable by zones, CRD with closure form zones is not a canonical representation of dense-time state-spaces. This is

---

Figure 1. examples of CRD and comparison with CDD
because CRD with closure form zones does not intrinsically have the capability to eliminate zones contained by the unions of other zones.

An alternative solution for space-efficient representation of state spaces is zones in their reduced form (called shortest-path reduction in [14]) which contains minimal number of clock difference constraints chosen by a policy. As shown in [14], DBM with zones in reduced form can be space-efficient. Wang has proposed to use CRD with zones in reduced form (or reduced CRD in short) as the representation for state-spaces to enhance verification efficiency [17]. Moreover, a space-efficient algorithm, with only four auxiliary variables, for computing reduced CRD has also been presented in [17].

Note again that reduced CRD is neither a canonical representation of dense-time state-space. Nevertheless, in average, CRD with zones in reduced form (reduced CRD in short) is much more space-efficient than previous data-structures [1, 7, 15, 16]. For example, given \( X = \{x_1, x_2, x_3\} \) and \( C_A = 10 \), in figure 1(b),(c),(d), we have the representations of zone \( \{0 - x_2 < -3, x_1 - x_3 \leq 1, x_2 - x_1 < -4\} \) in reduced CRD (figure 1(b)), in CRD with zones in closure form (figure 1(c)), and in CDD with zones in closure form (called tightened form in [7]) without FALSE terminal vertex (figure 1(d)). It is easy to see that as the number of clocks increases, reduced CRD will perform better and better.

3.2.3 Set-oriented manipulations on CRD. For convenience of discussion, given a CRD, we may just represent it as the set of zones recorded in it. Set-union (\( \cup \)), set-intersection (\( \cap \)), and set-exclusion (\( - \)) of two zone sets respectively represented by two CRDs are straightforward. For example, given CRDs \( D_1 : \{\zeta_1, \zeta_2\} \) and \( D_2 : \{\zeta_2, \zeta_3\} \), \( D_1 \cap D_2 \) is the CRD for \( \{\zeta_2\} \); \( D_1 \cup D_2 \) is the CRD for \( \{\zeta_1, \zeta_2, \zeta_3\} \); and \( D_1 - D_2 \) is the CRD for \( \{\zeta_1\} \). The complexities of the three manipulations are all \( O(|D_1| \cdot |D_2|) \).

Set-extraction (\( \mid \)) selects zones satisfying certain features from a zone set. Suppose \( D \) is the CRD for \( \{\zeta_1, \ldots, \zeta_k\} \) and \( D' \) is the CRD for \( \{\zeta_1, \ldots, \zeta_h\} \), then \( D \mid D' = \{\zeta_i \mid 1 \leq i \leq k; \exists 1 \leq j \leq h \forall x - x' \sim d \in \zeta_i(d \neq \infty \Rightarrow x - x' \sim d \in \zeta_i)\} \). The complexity is also \( O(|D_1| \cdot |D_2|) \).

Given two zones \( \zeta_1 \) and \( \zeta_2 \), \( \zeta_1 \ast \zeta_2 \) is a new zone representing the space-intersection of \( \zeta_1 \) and \( \zeta_2 \). Formally speaking, for every \( x, x' \) with \( \zeta_1(x, x') = (\sim_1, d_1) \) and \( \zeta_2(x, x') = (\sim_2, d_2), \zeta_1 \ast \zeta_2(x, x') = (\sim_1, d_1) \) if \( d_1 < d_2 \vee (d_1 = d_2 \wedge \sim_1 = "\sim") \), or \( \zeta_1 \ast \zeta_2(x, x') = (\sim_2, d_2) \) otherwise. Space-intersection (*) of two CRDs \( D_1 \) and \( D_2 \), in symbols \( D_1 \ast D_2 \), is a new CRD for \( \{\zeta_1 \ast \zeta_2 \mid \zeta_1 \in D_1, \zeta_2 \in D_2\} \). Our current implementation of the manipulation has complexity \( O(|D_1|^2 \cdot |D_2|^2) \).

3.2.4 CRD and BDD. It is possible to combine CRD and BDD into one data-structure for fully symbolic manipulation. Since CRD only has one sink vertex: true, it is more compatible with BDD without FALSE terminal vertex
which is more space-efficient than ordinary BDD. There are two things we need to take care of in this combination. The first is about the interpretation of default values of variables. In BDD, when we find a variable is missing during valuating variables along a path, the variable’s value can be interpreted as either TRUE or FALSE. But in CRD, when we find a variable for constraint \( x - x' \) is missing along a path, then the constraint is interpreted as \( x - x' < \infty \).

The second is about the interpretation of CRD manipulations to BDD variables. Straightforwardly, “∪” on Boolean variables is interpreted as “\(^{∨}\)” on Boolean variables. “∧” and “\(^{∧}\)” on Boolean variables are both interpreted as “\(^{∧}\)” on Boolean variables. \( D_1 - D_2 \) on Boolean variables is interpreted as \( D_1 \wedge \neg D_2 \) when the root variable of either \( D_1 \) or \( D_2 \) is Boolean. For \( D_1 \ast D_2 \), the manipulation acts as “\(^{∧}\)” when either of the root variables are Boolean. Due to page-limit, we shall omit the proof for the soundness of such interpretation.

From now on, we shall call it CRD+BDD a combination structure of CRD and BDD.

3.2.5 Variable ordering in CRD. In our BDD+CRD, we found the following evaluation ordering quite efficient in our experiment.

- All discrete variables precede those clock inequality variables in the evaluation ordering.
- Let \( 0 < x_1 < \ldots < x_n \). For clocks \( x, x', y, y' \in \{0, x_1, \ldots, x_n\} \), \( x - x' \) precedes \( y - y' \) iff either (1) \((x - y) \wedge (x - y') \lor (x' - y) \wedge (x' - y')\); or (2) \(x - y \lor x = y' \lor x' = y\).

Especially, the condition (2) of item 2 puts variable like \( x_2 - x_1 \) immediately below \( x_1 - x_2 \) and allows us to efficiently check for some trivial negative cycles.

4. CASCADE CRD

Reduced CRD indeed can be very space-efficient, as reported in [17], for some applications. But it is more difficult to efficiently decide the containment relation between two zones with reduced CRD. For example, we may have the reduced CRD in figure 2(a) for the state-space represented by formulæ (1) in the following.

\[
\begin{align*}
(x_1 - x_2 & \leq -2 \wedge x_2 - x_4 \leq -3 \wedge x_4 - x_1 \leq 5 \wedge x_1 = x_3) \\
\lor \quad (x_1 - x_2 & \leq -2 \wedge x_2 - x_4 \leq -3 \wedge x_4 - x_1 \leq 5)
\end{align*}
\] (1)

As can be seen that the zone of the first conjunction is actually contained in that of the second. This containment relation can be easily computed with all-pair shortest-path difference relation between clock pairs. But when we transform a given CRD into its reduced CRD, there is not enough information left to efficiently derive such containment relation. In specific, we cannot use the simple reduce operations described in the last paragraph of subsection 3.2.3 to eliminate zones which are contained by others in the same CRD. Thus, many
many zones may be represented in CRD for the reachable zone set which are actually subsets of zones already generated before. This problem may result in huge waste in both space and computation time. Actually, in our experiment, none of the web-camera implementations can be verified with reduced CRD.

Here we propose another representation scheme of CRD, called cascade CRD, which allows us to efficiently control CRD complexity with the simple reduction operations mentioned at the end of subsection 3.2.3. For the convenience of defining cascade CRD, we first define the cascade form of zones. A cascade CRD is merely a CRD, all whose zones are represented in their cascade form. Just like closure form and reduced form of zones, the cascade form of a zone is also a canonical representation of zone-characterizable state-spaces. The idea is to add a few more inequalities to zones in reduced forms to facilitate the decision of zone containment relation. Cascade CRD is especially designed for those systems which can have states in which many clocks have identical values. This kind of states may happen because of clock-reset operations at a synchronization, between a sender and a receiver, which is very common in modelling tightly-coupled interactions in concurrent systems.

In figure 2(b), we illustrate how we compute the cascade zone from its non-cascade zone represented by the first conjunction of formula (1), that is:

\((x_1 - x_2 \leq -2 \land x_2 - x_4 \leq -3 \land x_4 - x_1 \leq 5)\)
Given a zone $\zeta$, we use $\zeta^S$ to denote the cascade form of $\zeta$. The procedure to compute $\zeta^S$ from $\zeta$ can be presented as the following steps.

1. We classify the clocks in a state into equivalence classes, just as what has been done in [14]. Two clocks $x, x'$ are in the same equivalence classes in a zone $\zeta$ iff $x - x' = d \in \zeta^C$ for some $d \in N$. ($x - x' = d \in \zeta^C$ is a shorthand for $x - x' \leq d \in \zeta^C \land x' - x \leq -d \in \zeta^C$.) For example, in the first conjunction of formulus (1), all four clocks are in the same equivalence class; while in the second conjunction, only clock $x_1, x_2, x_4$ are in the same equivalence class.

In the following, then we divide the task into two subtasks. The first task is for adding inequalities in $\zeta^S$ between those clocks in the same equivalence classes while the second task is for adding inequalities in $\zeta^S$ between those clocks in different classes.

2. **Case of clocks in the same equivalence class:** Suppose we are given an equivalence class of clocks in a zone $\zeta$.

   (a) We first classify clocks in the given equivalence class into identity classes. Two clocks are in the same identity class if they have the same reading. For example, in the first conjunction of formulus (1), only clocks $x_1, x_3$ are in the same identity class and all other two clocks are not in the same identity class. But in the second conjunction, no two clocks are in the same identity class.

   (b) Then we arrange the identity classes in a linear sequence according to their less-than relation. For example, in the first conjunction of formulus (1), we have the following sequence.

   \[
   \{x_1, x_3\} \xrightarrow{-2} \{x_2\} \xrightarrow{-3} \{x_4\}
   \]

   Here the label of $-2$ is on the first “$\xrightarrow{}$” because $x_1 - x_2 \leq -2$ and $x_3 - x_2 \leq -2$. But in the second conjunction, the sequence for the equivalence class \{x_1, x_2, x_4\} is

   \[
   \{x_1\} \xrightarrow{-2} \{x_2\} \xrightarrow{-3} \{x_4\}
   \]

   (c) For each two clocks $x, x'$ in the same identity class, let $\zeta^S(x, x') = (\leq, 0)$.

   (d) For two clocks $x, x'$ such that the identity class of $x$ just precedes that of of $x'$ in the linear sequence, let $\zeta^S(x, x') = \zeta^C(x, x')$.

   (e) For a clock $x$ in the last identity class and a clock $x'$ in the first identity class, let $\zeta^S(x, x') = \zeta^C(x, x')$.

3. **Case of clocks not in the same equivalence class:** Suppose we are given clock $x, x'$ not in the same equivalence class. If $x$ is in the last identity
class of its equivalence class and \( x' \) is in the first identity class of its equivalence class, then let \( \zeta^S(x, x') = \zeta^C(x, x') \).

4. For any two inequivalent clocks \( x, x' \) such that \( (x, x') \) is redundant in \( \zeta \), we let \( \zeta^S(x, x') = (<, \infty) \).

5. For all other clock difference relation not covered in the above-mentioned steps, we let \( \zeta^S(x, x') = (<, \infty) \).

In the same style of the manipulation algorithm presented in [17], we can implement the above-mentioned procedure with symbolic CRD+BDD manipulation routines. But due to page-limit, we shall only present the above procedure. After CRD are stored with zones in cascade form (see for example in figure 2(c)), we can then use the reduction operation mentioned at the end of subsection 3.2.3 to efficiently eliminate many contained zones (see for example in figure 2(d)).

One good property of our cascade CRD is that for a zone \( \zeta \) without non-trivial identity classes (a class is trivial if it has only one element), then the corresponding zone in cascade form have the same number of inequalities as its counterpart in reduced form.

**LEMMA 1**: Given a zone \( \zeta \) such that no two clocks are in the same identity class in \( \zeta \), then \( \zeta^S \) and \( \zeta^R \) have the same number of inequalities of the form \( x - x' \sim d \) with either \( \sim \neq \neq < \) or \( d \neq \infty \).

**Proof**: When there is no nontrivial identity class, we will skip step 2(c) in the our procedure. This skip makes our procedure essentially identical to the one for constructing zones in reduced form presented in [14], except for the following. For an equivalence class with clock \( x \) and without clock \( x' \), an inequality like \( x - x' \sim d \) is called an incoming arc while one like \( x' - x \sim d \) is called an outgoing arc. In the zones in reduced form constructed according to [14], only incoming and outgoing arcs to the clocks in the first trivial identity class in the sequence can be kept. But in our cascade zones, outgoing arcs are recorded for the clocks in the first trivial identity class in the sequence while incoming arcs are recorded for the clocks in the last trivial identity class in the sequence. With this way of accounting, it is easy that both cascade form and reduced form of the same zone have the same number of clock inequalities.

Lemma 1 shows that our cascade CRD only adds to complexity when it is making effect on its target zones.

5. IMPLEMENTATION AND EXPERIMENTS

We have implemented our CRD-technology in version 3.0 of our tool red which was previously announced in [15, 16] (version 1.0, 2.0) and supports the modelling and safety-analysis of real-time systems with multiprocesses, pointer data-structures, and synchronizations (synchronous send and receive)
from one process to another. The new version, together with benchmarks, is now available at:

http://www.iis.sinica.edu.tw/~farn/red

Each process can use global and local variables of type clock, discrete, and pointer. Pointer variables either contain value NULL or the identifiers of processes. Thus in the models input to red, we allow complicate dynamic networks to be constructed with pointers.

At this moment, red supports backward reachability analysis. We have also implemented a reduction techniques in red. That is the reduction by elimination of inactive variables[13, 18] which is always executed. A variable is inactive in a state iff it is not read in any computation from the state before its content is overwritten. Contents of inactive variables can be omitted from state information without any effect on the computations.

We have tested our verifier with four implementations of the web-camera system, each with only one client. We have cooperated with Metamedia, a local company specializing in driver and peripheral software (http: //www.mmedia.com.tw), to test our new CRD-technology. One of their current projects is web cameras with browser interface (HTML language). The product supports multi-user connections, with web browsers, to dynamically monitor remote activities through internet. The software operates in application layer with an event queue and software interrupt handling. We want to analyze how long the event queue buffer needs to be.

In the following, we shall first describe how we model the system behavior, and then present our analysis result in section 5. There are three parameters in the system: number of clients (in symbols \#C), maximal length of queue (L), and the communication delay between server and clients [\beta, \alpha]. We have modeled implementations of the system with various values of \#C, L, and [\beta, \alpha]. The total number of processes we need is 3 + 2\#C. The processes are described in the following.

- The event queue which supports operations of dequeue and enqueue; and signalling of new event in the queue to the server. We need variables \(e_0, e_1, \ldots, e_{L-1}\) to records the content of the queue; variable \(l\) to records the current length of the queue.
- The camera process is also modeled by the single-mode automaton which outputs a video image every 20 time units. The video-ready event will be caught by the queue process throught a synchronizer.
- \#C timer processes for communication respectively with \#C clients. A timer process signals the first timeouts when its corresponding client does not respond in 40 time units. Then the server will send a new frame to the client. If still no response is received in 30 time units, the channel is disconnected.
client processes which acknowledges within time interval \([\beta, \alpha]\) the reception of a frame or dies.

• The server process which processes the events in the queue in the following way.
  – When the event is “VIDEO_READY” by camera, the server sends out a video-frame to each client. The processing time per client per video-frame is 5 time units. After processing the “VIDEO-READY” event, a timer is activated with timeout value 40 time units for each client.
  – When the event is “TIME_OUT\(_j\)” for client \(j\) at 40 time units, the server sends out a video-frame to client \(j\) again (in 5-time-unit processing time) and activates a timer with timeout value 30 time units.
  – When the event is “TIME_OUT\(_j\)” for client \(j\) at 30 time units, then the server thinks that the client is dead and disconnect the service to client \(i\).
  – When the event is “ACK\(_j\)” (acknowledgment) from client \(i\), then it disables the timer activated for client \(i\).

Two variables, each with possible values, orthogonally generate the four implementations. The first variable is \(L\), the event queue maximal length, with possible values of 1 and 2. The second variable is the response time interval from the client \([\beta, \alpha]\) with possible intervals of \([1, 1]\) and \([20, 40]\). The property to verify is whether the queue will overflow or not.

With reduced CRD, our tool cannot finish the verification task in 6 hours for any of the four implementations. But with cascade CRD, our tool is capable of coming up with the following performance data.

<table>
<thead>
<tr>
<th>([\beta, \alpha])</th>
<th>(L = 1)</th>
<th>(L = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 1])</td>
<td>no overflow/584s/812k</td>
<td>no overflow/2946s/3257k</td>
</tr>
<tr>
<td>([20, 40])</td>
<td>overflow/2320s/1378k</td>
<td>no overflow/3072s/2829k</td>
</tr>
</tbody>
</table>

s: seconds; k: kilobytes of memory in data-structure;

The performance data is collected on a Pentium II 366MHz with 256MB memory running Linux.

6. CONCLUSION

CRD gains its power primarily from intensive data-sharing. But its manipulations are usually burdened with high overhead. At this moment, CRD-technology only shows its edge when number of clocks is large (see the appendix). But we believe that as a new data-structure for the verification of timed automata, there can be much room for improvement on CRD. Especially, we will try out various zone forms and verification techniques appropriate for CRD.
REFERENCES


Appendix: Performance comparison between our CRD-based tool and Kronos and UPPAAL

The following table, from [17], shows performance data from Kronos, UPPAAL, and red w.r.t. implementations of benchmarks with various numbers of concurrent processes. Fischer’s mutual exclusion protocol are modified from [3, 13, 15, 18]. CSMA/CD is extracted from [19] while FDDI is from [5, 11]. Information on the three benchmarks can be found in http://www.iis.sinica.edu.tw/~farn/red.

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>concurrency</th>
<th>Kronos</th>
<th>UPPAAL</th>
<th>red no CRD</th>
<th>red (w. CRD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer’s mutual</td>
<td>3 processes</td>
<td>0.03s</td>
<td>0.014s</td>
<td>45.42s/1448k</td>
<td>1.12s/48k</td>
</tr>
<tr>
<td>exclusion</td>
<td>4 processes</td>
<td>0.14s</td>
<td>0.197s</td>
<td>871.2s/12703k</td>
<td>8.09s/162k</td>
</tr>
<tr>
<td></td>
<td>5 processes</td>
<td>0.989s</td>
<td>5.94s</td>
<td>O/M</td>
<td>48.6s/471k</td>
</tr>
<tr>
<td></td>
<td>6 processes</td>
<td>O/M</td>
<td>537.7s</td>
<td>O/M</td>
<td>278s/1271k</td>
</tr>
<tr>
<td></td>
<td>7 processes</td>
<td>O/M</td>
<td>O/M</td>
<td>O/M</td>
<td>1593s/3208k</td>
</tr>
<tr>
<td></td>
<td>8 processes</td>
<td>O/M</td>
<td>O/M</td>
<td>O/M</td>
<td>9352s/7788k</td>
</tr>
<tr>
<td>CSMA/CD</td>
<td>3 processes</td>
<td>0.032s</td>
<td>0.0046s</td>
<td>O/M</td>
<td>0.89s/60k</td>
</tr>
<tr>
<td></td>
<td>4 processes</td>
<td>0.071s</td>
<td>0.028s</td>
<td>O/M</td>
<td>3.63s/121k</td>
</tr>
<tr>
<td></td>
<td>5 processes</td>
<td>0.309s</td>
<td>0.216s</td>
<td>O/M</td>
<td>13.8s/278k</td>
</tr>
<tr>
<td></td>
<td>6 processes</td>
<td>1.915s</td>
<td>3.45s</td>
<td>O/M</td>
<td>60.6s/707k</td>
</tr>
<tr>
<td></td>
<td>7 processes</td>
<td>O/M</td>
<td>172s</td>
<td>O/M</td>
<td>227s/1781k</td>
</tr>
<tr>
<td></td>
<td>8 processes</td>
<td>O/M</td>
<td>O/M</td>
<td>O/M</td>
<td>709s/4426k</td>
</tr>
<tr>
<td></td>
<td>9 processes</td>
<td>O/M</td>
<td>O/M</td>
<td>O/M</td>
<td>3580s/10851k</td>
</tr>
<tr>
<td>FDDI token-ring</td>
<td>11 stations</td>
<td>399s</td>
<td>0.34s</td>
<td>N/A</td>
<td>1.98s/502k</td>
</tr>
<tr>
<td>passing</td>
<td>12 stations</td>
<td>O/M</td>
<td>0.487s</td>
<td>N/A</td>
<td>2.90s/591k</td>
</tr>
<tr>
<td></td>
<td>20 stations</td>
<td>O/M</td>
<td>4.05s</td>
<td>N/A</td>
<td>27.28s/1676k</td>
</tr>
<tr>
<td></td>
<td>30 stations</td>
<td>O/M</td>
<td>25.27s</td>
<td>N/A</td>
<td>151s/4165k</td>
</tr>
<tr>
<td></td>
<td>40 stations</td>
<td>O/M</td>
<td>95.99s</td>
<td>N/A</td>
<td>465s/7939k</td>
</tr>
<tr>
<td></td>
<td>50 stations</td>
<td>O/M</td>
<td>O/M</td>
<td>N/A</td>
<td>1107s/13513k</td>
</tr>
<tr>
<td></td>
<td>60 stations</td>
<td>O/M</td>
<td>O/M</td>
<td>N/A</td>
<td>1828s/21014k</td>
</tr>
</tbody>
</table>

Data collected on a Pentium III 800MHz with 256MB memory running LINUX; s: seconds; k: kilobytes of memory in data-structure; O/M: Out of memory; N/A: not available;

For the Fischer’s and CSMA/CD benchmarks, UPPAAL is invoked with options “-aSWD.” The performance data shows that CRD-technology is still more space-efficient and scales better w.r.t. number of clocks. For the FDDI benchmark, UPPAAL is invoked with options “-STDda” where “d” is for depth-first-search and suits very well for highly synchronous algorithms like FDDI.

We remind the readers that compared to those well-established tools like Kronos and UPPAAL. For example, UPPAAL can pass most of the benchmarks in very short time if the option of “convex-hull over-approximation” is switched on. But since our experiment was focused to argue that “with similar and equivalent verification techniques, BDD-like data-structures can outperform DBM,” we think it is better to just compare with Kronos and UPPAAL in the context of precise verification. Definitely, the success experience of Kronos and UPPAAL will be the guide for future enhancement of red. In the future, when such techniques are incorporated with CRD-based technology, we believe there will be much room for performance enhancement.
VERIFYING A SLIDING-WINDOW PROTOCOL USING PVS

Vlad Rusu
IRISA/INRIA, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France
rusu@irisa.fr

Abstract We present the deductive verification of safety and liveness properties of a sliding-window protocol using the PVS theorem prover. The protocol is modeled in an operational style which is close to an actual program. It has parametric window sizes for both sender and receiver, and unbounded, lossy communication channels carrying unbounded data. The proofs are done using invariant-strengthening techniques, encoded as PVS automated strategies based on heuristics and decision procedures.

1. INTRODUCTION

In the rapidly growing field of formal verification, deductive techniques [13] and tools [4, 8, 14] are general and powerful ways to verify properties of hardware and software systems. However, many practitioners and researchers hesitate to use deductive methods, which are perceived to be difficult, time-consuming and requiring advanced theoretical knowledge. This is in contrast with algorithmic techniques [2, 5, 9] which are completely automatic once a suitable abstraction of the system has been found and entered into, e.g., a model-checking tool.

The goal of this paper is to advocate the use of deductive techniques. Specifically, we show that although deductive verification is not fully automatic, it can be performed in a systematic way, using specific approaches, strategies, and automatic procedures on decidable subproblems that reduce the need for human intervention. This intervention remains necessary, but only as a general guideline to organize the verification process, i.e., to decompose the verification problem into manageable subproblems. Then, the subproblems can be solved in a highly automated manner.

As a case study we use an infinite-state sliding-window protocol for which we verify safety and liveness properties using the PVS theorem.
prover [14]. The protocol is a popular application for verification methods, both algorithmic and deductive. To our knowledge, our work is among the most general in terms of the properties verified and of the level of detail at which the protocol is modeled.

The rest of the paper is organized as follows. In Section 2 we describe the sliding-window protocol and related work on algorithmic and deductive approaches that have been applied to it. In Section 3 we briefly present the PVS theorem prover and describe, on a very simple example, the invariant-strengthening proof technique. In Section 4 we present in some detail our encoding of the sliding-window protocol in PVS, and in Section 5 we show how to prove safety and liveness properties by making extensive use of invariant strengthening. We conclude the presentation with ideas on how to further automate the deductive verification process. The specification and verification took three weeks to a moderately experienced PVS user and amounted to proving a total of eighteen theorems. PVS is available free of charge at http://pvs.csl.sri.com, and our sliding-window case study can be downloaded from http://www.irisa.fr/pampa/perso/rusu/forte.

2. RELATED WORK

We briefly present the sliding-window protocol and previous approaches that have been employed to verify it.

The Sliding-Window Protocol The sliding-window protocol (see, e.g., [17]) is a member of the data link layer. Its main function is to deliver a sequence of messages between a sender and a receiver in the correct order, regardless of possible losses that can occur during the transmission on the communication channels. The general principle underlying the protocol is that of pipelining. Instead of waiting for an acknowledgment after each message, the sender will continue sending messages until it has in its buffer, or “window”, a total of, say, $sw$ unacknowledged messages.

Similarly, the receiver will not just discard a message that has not the next expected sequence number, but will store it in its window, provided it is at most $rw$ many positions from the expected message. With this flexibility on both sides and using adequate values for the window sizes $sw$ and $rw$, typically computed from other parameters, e.g., the expected travel times of messages, the protocol can achieve very efficient transmission rates.

Verification of the Protocol The properties to be verified are divided into safety properties, which ensure that messages are delivered
in the correct order, and liveness properties, which guarantee that every message sent is eventually delivered.

**Algorithmic approaches.** These approaches typically verify finite-state instances of the protocol. In [15] safety properties for small window and channel sizes are verified by brute-force model-checking. In [12] the authors use one specific equivalence relation that allows them to handle window sizes as high as seven and to prove both safety and liveness properties. More recently, the paper [6] shows how to verify both kinds of properties on a version of the protocol with channel size eleven and window sizes two for both sender and receiver, using a specialized data structure called a Queue Difference Diagram.

**Deductive approaches.** These approaches typically verify general infinite-state specifications of the protocol, e.g., with unbounded window and channel sizes. An early deductive verification effort is presented in [1]. Only safety properties are verified, and a large number of invariants (over a thousand) were involved in the process. Another early work [3] presents the verification of safety and liveness properties of a specification of the protocol which includes real-time aspects. The level of automation of tools available at that time was low and the verification process included proving a large number (tens to hundreds) of trivial facts, e.g., that addition is commutative.

A recent paper involving a much more automated approach is [16]. Here, the authors use the weak monadic second-order logic with one successor (WS1S), a decidable logic that is expressive enough to encode finite sets and sequences and limited integer operations. A higher-level language is used to describe both the protocol and safety properties, which are then translated to WS1S. Two versions of the protocol are verified. One version considers a sender with an unbounded window size, which can be nondeterministically modified on-the-fly. In the second version, the sender’s window size is 256. The receiver’s window size is, in both versions, one. A total of eight invariants are required to prove the main safety property. Liveness properties are not considered.

**This work.** We verify an operational description of the protocol which is very close to a programming language, with parametric window sizes for both sender and receiver and unbounded communication channels. As [16], we emphasize on generality, automation, and compactness (i.e., there is a relatively small number of lemmas to prove), but treat the more general case of receiver’s window size greater than one and verify both safety and liveness properties.
3. PVS AND EXTENDED AUTOMATA

In this section we briefly describe the PVS specification and verification system and the model of extended automata that can be used for modeling reactive programs such as communication protocols. We demonstrate on a simple example how to prove properties of extended automata using the invariant-strengthening technique and how this technique is implemented in PVS. To the best of our knowledge, invariant strengthening in PVS was first mentioned in [11].

**PVS.** The PVS system consists of an input language, a typechecker, and an interactive prover. The input language is typed higher-order logic with a rich type system including simple types such as booleans, enumerations, integers, and records, and more complex function types, subtypes, dependent types, and abstract datatypes. Having such an expressive language makes it easy to specify, e.g., concurrent programs in a natural way, very close to a programming language. The drawback is that typechecking the input language is undecidable. However, PVS transforms this apparent weakness into an actual strength, because whenever the typechecker cannot decide whether an expression is type-correct, it generates a proof obligation. Most of the proof obligations can be discharged automatically, and obligations that cannot be proved often point to subtle errors in the specification.

A PVS proof is a tree, the root of which is the theorem being proved. The leaves of the tree are called pending subgoals. A proof proceeds as a sequence of commands, each of which transforms the proof tree by either proving a pending subgoal or by replacing a pending subgoal by a new set of pending subgoals. There are many proof commands, from propositional and first-order logic commands, to decision procedures and heuristic quantifier instantiation techniques, all of which can be combined into high-level, user-defined proof strategies.

**Extended Automata.** Extended automata are a computational model for reactive programs. An extended automaton $A$ consists of a finite set of typed variables $V$, an initial condition $\Theta$ and a finite set of guarded transitions $\mathcal{T}$. The variables can be either control or data variables; the control variables are of a finite type $\text{Location}$. Each transition $\tau \in \mathcal{T}$ is labelled and consists of a guard and an assignment. For example, in the automaton illustrated in Fig. 1, the type $\text{Location}$ consists of the two values $l_1, l_2$. There is one control variable $pc$ and one integer data variable $x$. The initial condition is $\Theta : pc = l_1 \land x = 0$. There are two
transitions, \( \tau_1 : l_1 \xrightarrow{inc} l_2 \) and \( \tau_2 : l_2 \xrightarrow{dec} l_1 \). The guard of transition \( \tau_1 \) is \( pc = l_1 \) and its assignment is \( x := x + 1 \).

![Extended Automaton Diagram]

**Figure 1.** Example of Extended Automaton

A *state* is a type-consistent valuation of the variables. Each transition \( \tau \) induces a *transition relation* \( \rho_\tau \) relating the possible before and after states. The global transition relation of the system is \( \rho_T = \bigcup_{\tau \in \tau_\rho} \rho_\tau \).

A *run* of the automaton is an infinite sequence of states, in which the first state satisfies the initial condition and every two consecutive states are in the transition relation. For example, in the extended automaton illustrated in Fig. 1, there is only one initial state that has location \( l_1 \) and variable \( x = 0 \). From this state the program can take the transition *inc* (its guard is *true*), assigning 1 to \( x \), moving to location \( l_2 \), etc.

An *invariant* is an assertion that holds at every state of every run. Invariants are the simplest form of safety properties. For example, it is not hard to show that \( x \geq 0 \) is an invariant of the automaton in Fig. 1.

**Invariant strengthening.** A state predicate is *inductive* if it holds initially and, if it is true at a given state \( s \), then it is also true at all states \( s' \) that are successors of \( s \) through the transition relation \( \rho_T \). For example, in the automaton represented in Figure 1, it is not hard to check that the predicate \( x \geq 0 \) is inductive. Indeed, it is true initially, and from any state satisfying \( x \geq 0 \), any transition, *i.e.*, *inc* or *dec*, will lead to a state also satisfying the predicate.

Clearly, an inductive predicate is an invariant. The converse is not true: consider, for example, the predicate \( pc = l_2 \supset x = 1 \). It is not hard to see that it is an invariant, that is, on every run, whenever control is at location \( l_2 \), variable \( x \) equals 1. But the predicate is not inductive: by knowing only that it is true before transition *inc*, it cannot be inferred that it is still true after the transition is taken. This does not mean the predicate is not an invariant, it is just that we cannot prove this fact by using only the predicate itself as induction hypothesis. To succeed
in the proof we need additional information, which can be obtained by invariant strengthening.

For example, consider again the predicate $\varphi : pc = l_2 \supset x = 1$, a state $s'$ of the automaton in Fig. 1 satisfying this predicate, and a state $s$ from which $s'$ is obtained by taking transition $inc$. Then, clearly, $s$ satisfies $\psi : pc = l_1 \supset x = 0$. The predicate $\psi$ is a pre-condition for $\varphi$ to hold. Now, it turns out that conjunction $\varphi \land \psi$ is inductive, thus, it is an invariant. In particular, we have just proved that $\varphi$ is an invariant too.

Thus, invariant strengthening is the following process: to prove a predicate is an invariant, first try to prove that it is inductive. If this is the case, then the proof is done. Otherwise, compute the pre-condition of the predicate, and try to prove that the conjunction of the invariant with the pre-condition is inductive. The process can be iterated until an inductive invariant is obtained. This is not guaranteed to happen in general, because the problem of proving invariants of general extended automata is undecidable. However, by using this approach in an interactive theorem prover, the user can often detect infinite patterns of behavior and formulate a predicate which is not just the pre-condition of a predicate by one transition, but the fixpoint of an infinite sequence of such operations. We demonstrate this latter feature in the next section.

4. THE SLIDING-WINDOW PROTOCOL IN PVS

In this section we present the sliding-window protocol in more detail, then show how to encode it as an extended automaton in PVS.

Architecture of the Protocol The architecture of the protocol is represented in Figure 2. The sender obtains data from a FIFO stream of data called the source. Each data element is first saved into the sender’s window, a FIFO buffer called sndWindow. The sender takes an element from its window, associates an index to it (a natural number keeping track of the order in which data has been obtained from the source), and sends the resulting record called a Message to the Message Channel MsgChan. The latter is a lossy FIFO, which may lose, but not reorder or create messages.

On the receiver side, messages are removed from MsgChan and, if a message’s index is within the bounds of the receiver’s window rcvWindow, it is stored there, otherwise, it is discarded. The receiver delivers contiguous sequences of messages to the external data target, and acknowledges the message with the highest index that it has delivered by sending the val-
ue of that index to the acknowledgment channel $\text{AckChan}$ (another lossy queue). Finally, the sender reads acknowledgments from $\text{AckChan}$. An acknowledgment is considered valid if it is between the bounds defined by the bottom of the sender’s window and the current top of that window, i.e., up to the message number obtained from the data source. A valid acknowledgement $a$ makes the bottom of the sender’s window become $a+1$, meaning that all messages with value at most $a$ are acknowledged.

**Figure 2.** Architecture of the Sliding-Window Protocol

**PVS encoding: declarations.** First, the *labels* of the protocol are encoded into theory `Action_and_types` (Figure 3) that includes some type declarations and an `Action` abstract datatype to encode the externally visible actions of the protocol. The meaning of each action is explained in the comments. Note that actions have *parameters*, e.g., `GET(d)` to express obtaining datum $d$ from the data source. (Technically, in terms of abstract datatypes, `GET` is a constructor, $d$ is an accessor, and `GET?` is a recognizer, but for better understanding it is preferable to see them as a declaration of actions with parameters.) Thus, the data being transferred between sender and receiver are natural numbers, and messages serve as recipients for the data together with an index (another natural number) that allows to reconstruct the order of messages, in case of losses during transmission.

The state of the automaton is encoded as the PVS record type `State` (cf. Fig. 4), with fields for variables of the sender, receiver, and channels:

Thus, the sender’s window size $sw$ and receiver’s window size $rw$ are strictly positive natural numbers. The receiver has three locations `receiving`, `flushing`, and `sendingAck`. Since there is only one location for the sender, it is not necessary to use a field for it. The meaning of the state variables is the following:

- variables for the data source: `source` is the infinite stream of data, modeled as a function from $\text{nat}$ to $\text{nat}$. `sourceIndex` is the index of the next data element that will be passed to the sender,

- variables for the sender: `sndWindow` is the sender’s window, modeled as a function from $\text{nat}$ to $\text{nat}$. `sndLow` is the bottom of the sender’s
Action_and_types : Theory
BEGIN
Msg: TYPE = [#data: nat, index: nat #]
Action: DATATYPE
BEGIN
GET(d: nat): GET? %get datum d from data source
SEND_MSG(m:Msg): SEND_MSG? %send message m to message channel
RCV_ACK(a:nat): RCV_ACK? %receive ack a from ack channel
TIMEOUT: TIMEOUT? %sender receives a timeout
RCV_MSG (m:Msg): RCV_MSG? %receive message m from message channel
PUT(d: int): PUT? %deliver datum d to data target
SEND_ACK(a: nat): SEND_ACK? %send ack a to ack channel
LOSE_MSG: LOSE_MSG? %environment loses message
LOSE_ACK: LOSE_ACK? %environment loses acknowledgment
END Action
END Action_and_types

Figure 3. Sliding-Window Protocol: Types and Actions

sliding_window: THEORY
BEGIN
IMPORTING Action_and_types
sw, rw : posnat % sender and receiver window sizes
rcvControlType: TYPE={receiving,flushing,sendingAck} %receiver location
State: TYPE =
[# source: [nat-> nat], sourceIndex : nat, % data source
sndWindow: [nat-> nat], %sender's window
sndlow, %bottom of sender's window
sndWinIndex, %counts data received from data source
sndIndex: nat, %counts data sent to message channel
rcvControl : rcvControlType, %control variable for the receiver
rcvWindow : [nat-> int], %receiver's window
cvlow, %bottom of receiver% window
headMsgChan,tailMsgChan: nat, msgChan: [nat->Msg], %message channel
headAckChan,tailAckChan : nat, ackChan: [nat->nat], %ack channel
target :[nat->int],targetIndex : nat, %data target
actual : boolean #] ... %rest of theory follows

Figure 4. Sliding-Window Protocol: State type

window, thus, at any moment in the execution, the current sender’s window ranges from sndWindow(sndLow) to sndWindow(sndLow+sw-1).
The variable sndWinIndex is the position in the sender’s window where the data source can place a data element, and variable sndIndex is the position of the next message that the sender will place of the message channel,

- variables of the receiver: rcvControl is the control variable. rcvWindow is the receiver’s window, and rcvLow is the bottom of the receiver’s
window. Thus, at any moment in time, the receiver’s window consists of elements from $\text{rcvWindow}(\text{rcvLow})$ to $\text{rcvWindow}(\text{rcvLow}+\text{rw}-1)$.

- variables of the channels: the message channel $\text{msgChan}$ is a function from $\text{nat}$ to $\text{Msg}$. It is a fifo whose active content at any moment in time ranges from $\text{msgChan}(\text{tailMsgChan})$ to $\text{msgChan}(\text{headMsgChan}-1)$. There are similar declarations for the acknowledgment channel $\text{ackChan}$.

**Initial states and transition relation.** The initial states, not shown here from lack of space, consist in setting every integer variable to zero, the receiver’s control to receiving, and receiver’s window to Absent (syntactic sugar for -1) to denote absence of any data in it: $\text{FORALL}(n: \text{nat}): s'\text{rcvWindow}(n) = \text{Absent}$.

The transitions of the model are encoded as a mapping next from State and Action to State. Action is the PVS abstract datatype depicted in Figure 3. There are nine transitions in all: four for the sender, three for the receiver, and two for the environment that may lose messages. The system works by nondeterministically firing an enabled transition, then choosing another transition and firing it, etc. A separate PVS theory runs.pvs (not shown here from lack of space) defines runs as sequences over a generic state type and the notions of invariants and inductive invariants.

The first transition of the sender specifies that a GET action with parameter $d$ can fire if there is space in the window: $s'\text{sndWinIndex} < s'\text{sndLow} + \text{sw}$, and $d$ is the next element of the data source. If this is the case, the sender’s window is modified to hold the value $d$ at position $\text{sndWinIndex}$, then, $\text{sndWinIndex}$ and $\text{sourceIndex}$ are both incremented:

```plaintext
GET(d) : IF s'\text{sndWinIndex} < s'\text{sndLow} + \text{sw} AND d = s'\text{source}(s'\text{sourceIndex}) THEN s WITH [\text{sndWindow}:= s'\text{sndWindow} \text{ WITH} [(s'\text{sndWinIndex}) := d], 
\text{sndWinIndex} := s'\text{sndWinIndex}+1, 
\text{sourceIndex} := s'\text{sourceIndex}+1] ELSE s WITH [\text{actual} := \text{false}] ENDIF,
```

The second transition of the sender takes a message $m$ whose data field is the element at position $\text{sndIndex}$ of the sender’s window, associates the value of that same index to it, places it on the message channel, and increments indices:

```plaintext
\text{SEND \_MSG}(m) : IF s'\text{sndIndex} < s'\text{sndWinIndex} AND m'\text{data}=s'\text{sndWindow}(s'\text{sndIndex}) AND m'\text{index} = s'\text{sndIndex} THEN 
 s WITH [\text{msgChan}:= s'\text{msgChan} \text{ WITH} [(s'\text{headMsgChan}) := m], 
\text{headMsgChan} := s'\text{headMsgChan}+1, 
\text{sndIndex} := s'\text{sndIndex}+1] 
```

A timer (not explicitly modeled in this specification) is started each time a SEND_MSG operation occurs. When a timeout occurs and there is no other action possible (i.e., sender’s window is full and all data from it has been sent to message channel and no acknowledgments are available), then prepare to start resending from bottom of sender’s window:

TIMEOUT: IF s'sndWinIndex = s'sndLow + sw AND s'sndIndex = s'sndWinIndex AND s'headAckChan = s'tailAckChan THEN
    s WITH [sndIndex := s'sndLow]
ELSE s WITH [actual := false] ENDIF,

The last action of the sender consist in receiving an acknowledgement a (if available) from the acknowledgment channel. If a is in the sender’s window, then we set the bottom of the sender’s window sndLow to a+1, meaning that all messages with index at most a have been acknowledged and will never be sent again. The index sndIndex (of the next message to be sent) is set to the maximum between itself and a+1:

RCV_ACK(a): IF s'headAckChan>s'tailAckChan AND a=s'ackChan(s'tailAckChan) THEN IF a >= s'sndLow AND a < s'sndWinIndex THEN
    s WITH[sndLow:= a+1,
    sndIndex:= IF a>s'sndIndex THEN a+1
    ELSE s'sndIndex ENDIF,
    tailAckChan := s'tailAckChan+1]
ELSE s WITH[tailAckChan := s'tailAckChan+1] ENDIF
ELSE s WITH [actual := false] ENDIF,

The first action of the receiver is to receive a message (if available) from the message channel. The control has to allow it (rcvControl = receiving). If the message’s index is in the window of the receiver, it is stored at the correct position:

RCV_MSG(m): IF s'rcvControl = receiving AND s'headMsgChan > s'tailMsgChan AND m = s'msgChan(s'tailMsgChan) THEN IF (m'index >= s'rcvLow AND m'index < s'rcvLow+rw) THEN
    s WITH[rcvWindow:=s'rcvWindow WITH[(m'index):=m'data],
    tailMsgChan:=s'tailMsgChan+1]
ELSE s WITH [tailMsgChan:=s'tailMsgChan+1] ENDIF
ELSE s WITH [actual:=false] ENDIF,

The second action of the receiver is to acknowledge data. The message with index just below the receiver window’s bottom is acknowledged. The control has to allow it (rcvControl = sendingAck), then control goes back to receiving:

SEND_ACK(a): IF s'rcvControl = sendingAck AND a = s'rcvLow-1 THEN
    s WITH[ackChan := s'ackChan WITH[(s'headAckChan) := a],
    headAckChan := s'headAckChan+1,
    rcvControl := receiving]
ELSE s WITH [actual := false] ENDIF,
The last action of the sender is also the most complex in the whole protocol. It consists in “flushing” the receiver window to the data target. This means sending to the data target the longest contiguous sequence (starting at the bottom of the sender’s window) of data items that are not equal to Absent. The \texttt{rcvControl} field is used to control this process. At the beginning, \texttt{rcvControl} = receiving, and if the bottom of the receiver window contains Absent, then the state is left unchanged, because there is nothing to flush. Otherwise, the control switches to flushing, and stays there to copy data from receiver’s window to data target until an Absent data item is met, in which case control goes to sending\_ack:

\begin{verbatim}
PUT(d) : IF (s'rcvControl = receiving OR s'rcvControl = flushing) AND
         d = s'rcvWindow(s'rcvLow) THEN
          IF d /= Absent THEN
            s WITH[r cvControl := flushing,
                   target := s'target WITH[(s'targetIndex):= d],
                   targetIndex:= s'targetIndex+1,rcvLow := s'rcvLow+1]
          ELSIF s'rcvControl =flushing THEN
            s WITH [rcvControl := sendingAck]
          ELSE s ENDIF
        ELSE s WITH [actual := false] ENDIF,
\end{verbatim}

Finally, there are two actions of the environment: losing a message and losing an acknowledgement (if there are any available). They consist in incrementing \texttt{tailMsgChan} and \texttt{tailAckChan}, respectively.

\section{Verifying Properties}

In this section we verify safety and liveness properties of the protocol.

\textbf{Safety Properties}

The main safety property required from the protocol is that the sequence of data delivered to the data target is a prefix of that obtained from the data source:

\texttt{main\_safety\_property: THEOREM invariant(LAMBDA (s: State):
    FORALL (i: nat): (i < s'rcvLow => s'target(i) = s'source(i)))}

The idea for proving Theorem \texttt{main\_safety\_property} is to prove the equality of sequences \texttt{source}, \texttt{target}, etc (cf. Fig. 2) from the “outside” to the “inside”. We prove, that up to a given position, the data source equals the sender’s window, the data target equals the receiver’s window, and that the two windows are equal. Then, the equality of these sequences imply the \texttt{main\_safety\_property} theorem:

- $\forall i < snd\text{WinIndex} : snd\text{Window}(i) = source(i)$
- $\forall i < rcv\text{Low} : rcv\text{Window}(i) = target(i)$
- \( \text{rcvLow} \leq \text{sndWinIndex} \)
- \( \forall i < \text{rcvLow} : \text{rcvWindow}(i) = \text{sndWindow}(i) \).

It should be noted that organizing the verification as above, \textit{i.e.}, decomposing the problem into four subproblems, is a user-dependent choice. However, once the subproblems are formulated as lemmas, the process of proving them is completely systematic. It is an invariant-strengthening process: a PVS strategy for proving inductiveness of invariants is applied. If the property is inductive, the proof succeeds, otherwise, PVS returns a number of pending subgoals. All these subgoals correspond to transitions that do not preserve the property under proof. By examining the subgoals, we formulate auxiliary invariants that, if proved, would eliminate the pending subgoals and settle the original property.

1: proving \( \forall i < \text{sndWinIndex} : \text{sndWindow}(i) = \text{source}(i) \). This is expressed in PVS as Lemma \text{source_equals_send} below. By applying our PVS strategy for proving inductive invariants, we obtain two pending subgoals, which suggest to prove the auxiliary Lemma \text{source_equals_send_1}:

\[
\text{source_equals_send} : \text{LEMMA invariant}(\text{LAMBDA}(s: \text{State}): \text{FORALL}(i: \text{nat}): i < s'\text{sndWinIndex} \Rightarrow s'\text{source}(i) = s'\text{sndWindow}(i))
\]

\[
\text{source_equals_send_1} : \text{LEMMA invariant}(\text{LAMBDA}(s: \text{State}): s'\text{sndWinIndex}=s'\text{sourceIndex})
\]

The conjunction of the predicates in the above lemmas is inductive, and the strategy for proving inductive invariants succeeds: the step is completed.

2: proving \( \forall i < \text{rcvLow} : \text{rcvWindow}(i) = \text{target}(i) \). This is similar to Step 1 and is expressed in PVS as Lemma \text{target_equals_receive} below. The approach for proving it also takes one invariant-strengthening step.

\[
\text{target_equals_receive} : \text{LEMMA invariant}(\text{LAMBDA}(s: \text{State}): \text{FORALL}(i: \text{nat}): i < s'\text{rcvLow} \Rightarrow s'\text{target}(i) = s'\text{rcvWindow}(i))
\]

3: proving \( \text{rcvLow} \leq \text{sndWinIndex} \). This step and the following are more complex and we explain them in more detail. The property is expressed as:

\[
\text{rcvLow_leq_sndWinIndex} : \text{LEMMA invariant}(\text{LAMBDA}(s: \text{State}) : s'\text{rcvLow} <= s'\text{sndWinIndex})
\]

The inductiveness strategy fails on the above lemma. The pending subgoal in the PVS proof identifies the PUT transition of the protocol (cf. Section 4) to be responsible for the failure. This is because the PUT transition increments variable \text{rcvLow}, thus, if \text{rcvLow} \leq \text{sndWinIndex} holds before the transition is taken, it might not hold afterwards. The subgoal also suggests how to solve the problem: by showing (1): \text{rcvWindow}(

\text{sndWinIndex} \)}
Absent. This is because the PUT transition can increase rcvLow and invalidate Lemma \( rcvLow_{\leq} sndWinIndex \) only when both \( rcvLow = sndWinIndex \) and \( rcvWindow(rcvLow) \neq Absent \) hold before the transition. By proving (1), we prove that the latter situation cannot happen.

Now, if we formulate (1) as a lemma and try to prove it by basic invariance strengthening we run into an infinite loop. Indeed, PVS requires to prove \( rcvWindow(sndWinIndex+1) = Absent, \) \( rcvWindow(sndWinIndex+2) = Absent, \) etc. Noticing this pattern we realize that the actual lemma we need to prove is the “fixpoint” of these properties:

\[
rcvLow_{\leq} sndWinIndex_1: \text{LEMMA} \text{ invariant}(\lambda s: \text{State}: \forall i: \text{nat}: i \geq s'sndWinIndex \Rightarrow s'rcvWindow(i) = Absent)
\]

We apply the inductiveness strategy to prove this lemma, and fail again. The pending subgoal now points to transition RCV_MSG of the protocol (cf. Section 4). Indeed, this transition modifies rcvWindow, which could invalidate the \( rcvLow_{\leq} sndWinIndex_1 \) lemma. There is one pending subgoal, which suggests to prove the property (2) \( \text{msgChan}(\text{tailMsgChan}) \ 'index < sndWinIndex. \) Indeed, in this case \( rcvWindow \) is modified at a position that is not referred to in the above lemma, thus, the lemma will not be invalidated if (2) holds. But attempting to prove (2) by simple invariance-strengthening actually points us to proving the more general lemma

\[
rcvLow_{\leq} sndWinIndex_2: \text{LEMMA} \text{ invariant}(\lambda s: \text{State}: \forall i: \text{nat}: i < s'headMsgChan \Rightarrow s'msgChan(i)'index < s'sndWinIndex)
\]

The inductiveness strategy succeeds in proving this lemma, and Step 3 is done.

4: proving \( \forall i < rcvLow : rcvWindow(i) = sndWindow(i). \) This property is expressed as the following PVS lemma:

\[
\text{send\_equals\_receive: \text{LEMMA} \text{ invariant}(\lambda s: \text{State}: \forall (i : \text{nat}): i < s'rcvLow \Rightarrow s'sndWindow(i) = s'rcvWindow(i))}
\]

The inductiveness strategy fails to prove the lemma, and leaves two pending subgoals. The first subgoal corresponds to the GET transition of the protocol (cf. Section 4). This is because that transition modifies sndWindow, which could invalidate the lemma. The second subgoal corresponds to the PUT transition, which modifies rcvLow and could also invalidate the lemma.

To prove the first subgoal it is enough to prove (3) \( rcvLow \leq sndWinIndex. \) Indeed, the GET transition modifies sndWindow at position sndWinIndex, and (3) would guarantee that this modification has no influence on our lemma. We have already proved (3) in Lemma \( rcvLow_{\leq} sndWinIndex. \)

The second subgoal suggests to prove: if (4) \( rcvWindow(rcvLow) \neq Absent, \) then (5) \( rcvWindow(rcvLow) = sndWindow(rcvLow). \) This is because if (4) holds, then the PUT transition increments rcvLow, and what we have
to show is that the predicate in Lemma send_equals_receive holds when $i = rcvLow$, which is implied by (5).

Now, proving that (4) implies (5) by simple invariant strengthening would again lead us into an infinite loop, thus, the solution is to prove

$$\text{send_equals_receive}_1: \text{LEMMA invariant} (\lambda s: \text{State}: \forall i: \text{nat}: (i > s'rcvLow \land s'rcvWindow(i) /=Absent) \Rightarrow s'rcvWindow(i) = s'sndWindow(i))$$

This property is not inductive, and the strategy fails again, leaving two pending subgoals. The first subgoal is due to transition GET, which modifies sndWindow, and can be settled by proving that $rcvWindow(sndWinIndex) = Absent$ always holds before this transition. This was already proved in Lemma $rcvLow_leq_sndWinIndex_1$. Finally, the second subgoal is due to transition RCV_MSG modifies rcvWindow, and is settled by proving the following lemma, which is inductive and is proved automatically.

$$\text{send_equals_receive}_2: \text{LEMMA invariant} (\lambda s: \text{State}: \forall i: \text{nat}: i < s'headMsgChan \Rightarrow s'msgChan(i)'data = s'sndWindow(s'msgChan(i)'index))$$

**Liveness Properties**

The main liveness property is that every data item from the data source is eventually delivered to the data target. If we call fair the runs that eventually obtain every data item from the source, and live the runs that eventually deliver every data item to the target, then we have to show that every fair run is live:

$$\text{fair} : \text{PRED}[(\text{run})] = \lambda (r: (\text{run})) : \forall m: \text{nat}: \exists n: \text{nat}: r(n)'sourceIndex > m$$

$$\text{live} : \text{PRED}[(\text{run})] = \lambda (r: (\text{run})) : \forall m: \text{nat}: \exists n: \text{nat}: r(n)'targetIndex > m$$

$$\text{main_fairness_property: THEOREM FORALL}(r:(\text{run})): \text{fair}(r) \Rightarrow \text{live}(r)$$

Here, $(\text{run})$ is the type of runs, defined in theory runs.pvs as the subtype of function from nat to State whose initial state satisfy the initial condition and such that any two consecutive states satisfy the transition relation. As for safety, the proof of Theorem main_fairness_property is divided into several steps and required to prove auxiliary invariants.

$$\text{fair_aux1}: \text{PRED}[(\text{run})] = \lambda (r: (\text{run})): \forall m: \text{nat}: \exists n: \text{nat}: r(n)'sndWinIndex > m$$

$$\text{fairness_property_aux1: LEMMA FORALL}(r:(\text{run})): \text{fair}(r) \Rightarrow \text{fair_aux1}(r)$$

$$\text{fair_aux2}: \text{PRED}[(\text{run})] = \lambda (r: (\text{run})): \forall m: \text{nat}: \exists n: \text{nat}: r(n)'sndLow > m$$

$$\text{fairness_property_aux2: LEMMA FORALL}(r:(\text{run})): \text{fair_aux1}(r) \Rightarrow \text{fair_aux2}(r)$$

$$\text{safe_aux2}: \text{LEMMA invariant} (\lambda s: \text{State}: s'sndWinIndex <= s'sndLow + sw)$$

$$\text{safe_aux3}: \text{PRED}[(\text{run})] = \lambda (r: (\text{run})): \forall m: \text{nat}: \exists n: \text{nat}: r(n)'rcvLow > m$$

$$\text{safe_aux3bis : LEMMA invariant} (\lambda s: \text{State}: \forall i: \text{nat}: i < s'headAckChan \Rightarrow s'ackChan(i) < s'rcvLow)$$

$$\text{safe_aux3 : LEMMA invariant} (\lambda s: \text{State}: s'sndLow <= s'rcvLow)$$
6. CONCLUSION AND FUTURE WORK

In an attempt to demonstrate that deductive methods can be used in a systematic way to validate communication protocols, we present the verification of safety and liveness properties of a sliding-window protocol using the PVS specification and verification system. Some final remarks in favor of our thesis:

- The proofs of the safety properties are almost all the same. Indeed, most of the new proofs during the verification process were created by copy and paste, and only some quantifiers needed specific intervention to be properly instantiated. This was also true to some extent for the liveness properties,

- the strategy for proving inductive invariants suggests, in case of failure, new invariants to prove for settling the unproved subgoals. Thus, the method provides useful feedback for performing invariant strengthening,

- the new invariants did not need to be carefully analyzed and fully understood for proceeding in the proof. We have found the mechanized proof with PVS to be an automatic process, which does include the human in the loop, but does not excessively tax his patience or ingenuity.

We are currently working on techniques to further the automation of deductive verification. In particular, we are interested in a class of extended automata with integer variables and function symbols for which we study the automatic generation of auxiliary invariants and the automatic proof of inductiveness of invariants in the context of the automatically generated invariants. The class is expressive enough to cover the sliding-window protocol specification, and a large fraction of the properties presented in this paper.

Acknowledgment

Thanks to Duncan Clarke for carefully reading the manuscript and for useful comments and suggestions.
REFERENCES


Part Six

Test Sequence Derivation
This page intentionally left blank.
TEST SEQUENCE SELECTION

David Lee and Ruibing Hao
Bell Labs Research China, Beijing, China

Abstract We study the test sequence selection problem. Given a large number of tests for a protocol system, we want to select a subset of tests so that the number of tests is reduced yet the test coverage is not sacrificed. We discuss the complexity of the test selection problem and propose a class of algorithms for different protocol system information requirements, test coverage criteria, and cost. We report experimental results on ISDN connection control protocol system Q.931.

Keywords: protocol system, test sequence, test coverage, extended finite state machine, reachability graph, NP-hard, algorithm, tree, sorting

1. INTRODUCTION

With advanced computer technology protocol systems are getting larger to fulfil complicated tasks. However, they are also becoming less reliable. Testing is an indispensable part of system design and implementations; yet it has proved to be a formidable task for complex systems. Because of its practical importance and theoretical interest, there have been a lot of activities on protocol system testing. There are conformance testing, interoperability testing, and performance testing. Conformance testing is to test conformance of system implementations to their specifications [Br, LY1, PBG]. Interoperability testing tends to uncover faults when different system components are interoperating or interfacing with each other [GHLS, KK]. Conformance and interoperability testing are designed to check the correctness of system behaviors whereas performance testing is related to the system performance, such as its transmission rate.
For conformance testing, test sequences are generated and then applied to the systems under test to conclude whether the implementations conform to their specifications. For the conducted tests to be conclusive, we have to generate test sequences with desired fault coverage. Furthermore, it is costly to execute tests, and we want to minimize the number of tests yet without sacrificing the fault coverage.

For certain complicated legacy protocol systems, such as 5ESS (AT&T/Lucent No. 5 Electronic Switching System), tests have been generated and applied to the systems over the years at different development stages and by different test engineers. There are thousands of test sequences in the available test set. To test such systems, it is impractical to generate tests from scratch, since it is often impossible to model the systems because they are too complex. Therefore, we want to use the available test set accumulated over years. However, we do not want to execute all of them, since there are too many and to run each test takes a substantial amount of time. A natural solution in practice is to select test sequences among the available test set.

The test selection problem was studied in [LPB] based on the valuations of the test sequences to be selected, and it was reduced to an optimisation problem over Boolean algebra. For testing in context, the problem was studied in [YCL]. An important issue of test selection is the possible loss of coverage. This problem was investigated in [LPB] for partially specified machines and also in [CV, VC, ZV] with an elegant metric of coverage. The selection criteria and their notations were studied in [Pa].

In this paper, we study the following test selection problem. We have a large set of test sequences, and we want to select a minimal subset of tests to execute without sacrificing the fault coverage. For a formal study, we use extended finite state machine to model protocol systems. It is only for convenience. Our results are easily adaptable to other equivalent mathematical models such as labeled transition systems. We discuss the complexity of the problem and propose efficient algorithms for the test sequence selection.

In Section 2 we describe an extended finite state machine model and its reachability graph. In Section 3 we formulate a problem of test sequence selection, discuss the coverage criteria, and study the problem complexity. In Section 4 we discuss efficient algorithms for the test selection. In Section 5 we discuss a more general problem when the reachability graph is not available. In Section 6 we report experimental results on ISDN Q.931 system
testing. In Section 7 we conclude the paper with a comparison of all the algorithms studied.

2. BASICS

A finite state machine contains a finite number of states and produces outputs on state transitions after receiving inputs. It is often used to model control portions of protocol systems. However, data portions of protocols include variables and operations based on their values; ordinary finite state machines are not powerful enough to model in a succinct way the physical systems any more. Extended finite state machines (EFSM), which are finite state machines extended with variables, have emerged from the design and analysis of communication protocols [LY1]. For instance, IEEE 802.2 LLC [ANSI] is specified by 14 control states, a number of variables, and a set of transitions (pp. 75-117). For a formal definition of EFSM and the related concepts, see [LY1].

Each combination of a state and variable values is called a configuration. An EFSM usually has an initial state $s_0$ and all the variables have an initial value $x_{\text{init}}$; we have the initial configuration $(s_0, x_{\text{init}})$. A reachability graph consists of all the configurations and transitions, which are reachable from the initial configuration. It is a directed graph where the nodes and edges are the reachable configurations and transitions, respectively. Obviously, a control state may have multiple appearances in the nodes (along with different variable values) and each transition may appear many times as edges in the reachability graph.

For a path from the initial node (configuration) of the reachability graph, the input/output (I/O) labels on the transitions of the path provide an I/O sequence. Conversely, if an I/O sequence corresponds to a unique path from the initial node in the reachability graph, then the underlying EFSM is deterministic. Otherwise, it is non-deterministic. For clarity, in this paper we only consider systems, which are modelled by deterministic EFSM’s. Our approaches can be modified to handle non-deterministic EFSM’s, as we shall comment in the conclusion of the paper.

Given an input sequence, there is at most one path from the initial node in the reachability graph such that the transitions on the path are labelled with the input sequence, since the machine is deterministic. If such a path exists, then the given input sequence is valid. Otherwise, it is invalid.
Obviously, a valid input sequence corresponds to a unique I/O sequence, which are the labels on the corresponding unique path from the initial node.

We first study the test selection problem, assuming that the underlying system reachability graph is available. We then relax the problem with an assumption that only the EFSM specification of the underlying system is available. Finally, we present algorithms for a general case when no information of the underlying system is available for test selection.

3. TEST SEQUENCE SELECTION PROBLEM AND ITS COMPLEXITY

A test sequence (or a scenario) is valid input sequence of a protocol system that is modelled by an EFSM. Since the system is deterministic, there is a one-to-one correspondence between test sequences and paths in the reachability graph from the initial configuration. In practice, a test sequence usually consists of an I/O sequence; the input sequence is applied to the system under test, and the output sequence is to be observed from the system response. Therefore, a test sequence can be represented by a valid input sequence, a valid I/O sequence, or a path from the initial configuration in the reachability graph of the underlying protocol system. For convenience, we shall use these terms interchangeably.

Informally, the test sequence selection problem is: Given a set of test sequences \( S \), select from \( S \) a subset of tests \( S^* \) with a desirable fault coverage. Fault coverage is essential for testing. However, it is often specified differently from different system models and practical needs.

For conformance testing of FSM's we may consider checking sequences [LY1, Mo], which guarantee that an implementation machine is structurally isomorphic to the specification machine. For FSM's there are polynomial time algorithms for constructing the checking sequences, however, even for medium size machines it is too long to be practical [YL1]. For EFSM's hundreds of thousands of states (configurations) are typical and it is in general impossible to construct checking sequences. For the test selection problem, if we are interested in checking sequences, the problem is even harder. Specifically, given a set of tests \( S \), we want select a subset \( S^* \) from \( S \) such that \( S^* \) are checking sequences. The problem is NP-hard [Ya].

A commonly used criterion of fault coverage is to test each edge in the reachability graph at least once. It has been a well-accepted criterion in
practice, and we first consider this criterion. Formally, given an EFSM $M$, let $G$ be its reachability graph with an initial configuration $v_0$, which corresponds to the initial state of $M$ and the initial variable values. A test sequence of $M$ is a path in $G$ from $v_0$. A covering test set is a set of test sequences such that each edge of $G$ is covered by at least one of the test sequences. We want to select a subset of tests such that it is still a covering test set and contains a minimal number of tests:

**Problem 1. Test selection with a reachability graph.** Given the reachability graph of an EFSM and a covering test set $S$, select a subset of test sequences $S^*$ from $S$, such that $S^*$ is a covering test set with a minimal cardinality.

The problem is hard. We can reduce the set cover problem to this test selection problem. Since the set cover problem is NP-hard [GJ], we have:

**Proposition 1.** The test selection problem 1 is NP-hard.

Therefore, it is hard to obtain an optimal solution for test selection in general. We discuss heuristic methods next.

4. **TEST SEQUENCE SELECTION WITH A REACHABILITY GRAPH**

We consider the following greedy method. We first find a test sequence in $S$ that covers a maximal number of uncovered edges of the reachability graph, and add this test to $S^*$. We then repeat the process until all the edges of the reachability graph are covered. Formally,
Algorithm 1.

input: reachability graph $G$ and a covering test set $S$
output: a covering test set $S^*$, which is a subset of $S$

1. mark all edges in graph $G$ as uncovered;
2. $S^* = \emptyset$; /* an empty set */
3. repeat
4. find a test sequence $s$ in $S$ that covers a maximal number of uncovered edges of the graph $G$, and mark these edges as covered;
5. $S = S - \{s\}, S^* = S^* \cup \{s\}$;
6. until all edges in $G$ are covered
7. return $S^*$

Figure 1. Algorithm 1

It takes polynomial time to find a test sequence $s$ from $S$ that covers a maximal number of uncovered edges. A straightforward approach is to examine each test sequence in turn, and it takes time proportional to the length of the test sequence examined. Suppose that there are $N$ test sequences in $S$ and the total length of all the test sequences in $S$ is $M$. Then it takes time $O(M)$ to select a test and time $O(MN)$ in the worst case to determine the test set $S^*$.

Proposition 2. Algorithm 1 takes time $O(MN)$ to select a covering test set where $M$ and $N$ are the total length and number of all the test sequences in the given covering test set.

Algorithm 1 provides a heuristic approach, that is, the selected test set is a covering set but may not be minimal. Instead of selecting one test at a time, we could select two tests at a time such that their combined coverage is maximal [C8]. It increases the cost but still runs in polynomial time. On the other hand, it may reduce the number of selected tests. In general, can we do better than Algorithm 1 in polynomial time? With a similar argument as in [LY2] it can be proved that we cannot do better than a logarithmic factor in polynomial time. That it, for any polynomial time algorithms, which select a covering set of cardinality $N^*$, there are cases such that $N^* = \Omega(N^{**} \log N^{**})$ where $N^{**}$ is the cardinality of a minimal covering test set, unless of course $P=NP$.

It takes time $O(MN)$ for Algorithm 1 to select a covering test set where $M$ and $N$ are usually large in practice. To reduce the cost we may want to
avoid examining all the tests in the set. We can conduct the following preprocessing and variations of Algorithm 1.

**Algorithm 1.1.** We first sort the tests in S according to their lengths in a descending order, and then examine them in that order.

The rationale is: longer tests correspond to longer paths in the reachability graph and may cover more edges of the graph. Furthermore, if we have found a test, which covers k uncovered edges, then there is no need to examine tests of length k or less. Since there are N tests in the set, the added cost of sorting is $O(N \log N)$, which is negligible since it is dominated by $O(MN)$.

**Algorithm 1.2.** Select each test sequence to be examined from S uniformly at random.

As often observed in practice, randomization helps. See the experimental results in Section 6.

**Algorithm 1.3.** Sort the test sequences by their input symbols alphabetically rather than their lengths, remove all the tests which are a proper prefix of other tests, and then apply Algorithm 1.

Obviously, this preprocessing reduces redundant tests and we have fewer tests to process. We shall further discuss this variation in the next section.

## 5. TEST SEQUENCE SELECTION WITHOUT A REACHABILITY GRAPH

Often we only have a test set available but do not have the corresponding reachability graph of the underlying system or it is too costly to construct such a graph. The problem becomes harder since we do not know whether a selected test set covers all the edges in the reachability graph.

Suppose we have the underlying system specification EFSM yet without its reachability graph. We have:

**Problem 2. Test selection with an EFSM.** Given an EFSM and a covering test set S, select a subset of test sequences $S^*$ from S, such that $S^*$ is a covering test set with a minimal cardinality.
Similar to Problem 1, we have:

**Proposition 3.** The test selection problem 2 is NP-hard.

Therefore, it is hard to obtain an optimal solution for test selection in general. On the other hand, the remarks on the complexity of Problem 1 also apply to Problem 2.

We can use the following heuristic procedure. We examine all the test sequences in \( S \), trace each sequence in the EFSM, and record all the edges covered in the corresponding reachability graph. We then select test sequences in \( S \) until all the recorded edges are covered. The selection procedure is similar to Algorithm 1, and the variations of Algorithm 1 also apply here. Note that we do not construct the reachability graph explicitly.

**Algorithm 2.**

*input:* EFSM \( M \) and covering test set \( S \)

*output:* a covering test set \( S^* \), which is a subset of \( S \)

1. \( E = \emptyset \); /* identify all edges of reachability graph; \( E \) is an empty edge set */
2. **for** each test sequence \( s_i \) in \( S \), \( i = 1, \ldots, N \)
3. \( u = u_0 = s_0 x_{init} \); /* \( u_0 \) is the initial configuration of \( M \) */
4. **for** \( j = 1, \ldots, k_i \); /* \( k_i \) is the number of transitions in \( s_i \) */
5. trace transition \( t_{ij} \) in \( M \) and determine next configuration \( v \);
6. **if** corresponding edge in reachability graph \((u,v) \notin E\)
7. \( E = E \cup \{(u,v)\} \);
8. \( u = v \);
9. set all edges in set \( E \) as uncovered; /* find a covering subset \( S^* \) */
10. \( S^* = \emptyset \); /* an empty set */
11. **repeat**
12. find a test sequence \( s \) in \( S \) that covers a maximal number of uncovered edges in set \( E \), and mark these edges as covered;
13. \( S = S - \{s\}, S^* = S^* \cup \{s\} \);
14. **until** all edges in \( E \) are covered
15. **return** \( S^* \)

**Figure 2.** Algorithm 2

Given the underlying EFSM and test set \( S \), Algorithm 2 first collects all the edges of the corresponding reachability graph of the EFSM, and then selects a covering subset of tests. One could construct a reachability graph of
the EFSM and then select tests using Algorithm 1. However, it is often a difficult task to explicitly construct a reachability graph due to the state explosion. Algorithm 2 avoids this process.

Algorithm 2 traces all the tests in Line 1-8 and records the edges in $E$. The total number of edges in the worst case can be proportional to $M$. Therefore, it takes time $O(M)$ to check if an edge under consideration is in $E$, and the total time is $O(M^2)$. We can keep the recorded edges in set $E$ in a sorted order (by their inputs, the start and end states and their associated variable values). It takes time proportional to $O(\log M)$ to check if an edge is in $E$, and the total time to execute Line 1-8 is $O(M\log M)$. By a same argument as for Algorithm 1, the run time of Line 9-15 is $O(MN\log M)$. In summary,

**Proposition 4.** Algorithm 2 takes time $O(MN\log M)$ to select a covering test set where $M$ and $N$ are the total length and number of all the test sequences in the given covering test set.

Algorithm 2 is for a more general test selection Problem 2 where there is no reachability graph and also no need to construct one. However, there is an extra price to pay in terms of the run time, i.e., a factor of $\log M$.

Often in practice the underlying system model may not be available, especially for those legacy systems, and we only have a set of tests to select. This is the most general case of the test selection problem:

**Problem 3. Test selection without any information of the underlying protocol system.** Given a covering test set $S$ for a protocol system, for which there is no information available, select a subset of test sequences $S^*$, such that $S^*$ is a covering test set.

Different from Problem 1 and 2, in general it is impossible to select a minimal covering test set since there is no information of the underlying protocol system.

For this problem we only consider the following redundancy criterion. Let $s$ and $s^*$ be two test sequences where $s^*$ is a prefix of $s$. Let $p$ and $p^*$ be the corresponding paths from the initial configuration in the reachability graph, which we do not know. Since the corresponding EFSM is deterministic, $p^*$ is a prefix of $p$ as paths in the reachability graph, and we can remove $s^*$ from the test set without sacrificing the coverage. Consequently, if $S$ is a covering test set, the test subset $S^*$, selected by discarding all the prefixes, is still a covering test set. As a matter of fact, in
the worst case, we cannot reduce the tests anymore. Consider the following reachability graph. It consists of separate paths from the initial node. For test selection, we can only discard the paths (tests), which are a proper prefix of another path; any further reduction will lose the coverage.

The above observations lead to the following algorithm. Consider the alphabet set of all the input symbols. We sort all the input sequences in \( S \) alphabetically and then remove all the input sequences (test sequences), which is a prefix of another input sequence. To check whether a test sequence \( s_i \) is a prefix of another test, there is no need to check \( s_i \) again all the tests in \( S \); since the test set is sorted, we only need to check \( s_i \) with the next test sequence \( s_{i+1} \).

**Lemma 1.** A Test \( s_i \) in \( S \) is a prefix of another test if and only if it is a prefix of \( s_{i+1} \).

**Proof.** We only need to prove that if \( s_i \) is a prefix of another test \( s_i t \) then it is also a prefix of \( s_{i+1} \). Since \( s_{i+1} \) is sorted in between \( s_i \) and \( s_i t \), it must be of a form \( s_i r \). Therefore, \( s_i \) is a prefix of \( s_{i+1} = s_i r \).

**Algorithm 3.**

*input: test sequence set \( S \) and alphabet set of all the input symbols \( \Sigma \)  
output: subset \( S^* \) of \( S \) with redundant tests removed*

1. sort all input sequences of \( S \) alphabetically;
2. \( S^* = \Lambda; \) /* an empty set */
3. for \( i = 1, ..., N-1 \) /* \( N \) is the cardinality of \( S^* \) */
4.  if \( s_i \) is not a prefix of \( s_{i+1} \)
5.  \( S^* = S^* \cup \{s_i\} \);
6.  \( S^* = S^* \cup \{s_N\} \);
7.  return \( S^* \)

**Figure 3.** Algorithm 3

Since we only remove the redundant tests, we have:

**Proposition 5.** Given a test set \( S \), Algorithm 3 removes redundant tests and selects a subset of tests \( S^* \). If \( S \) is a covering test set of the reachability graph of the underlying protocol system, then \( S^* \) remains a covering test set.

Note that Algorithm 3 may select test sequences that are redundant in terms of covering the reachability graph since we do not have its information.
Therefore, the resulting number of tests may be larger than that from Algorithm 1 and 2. See Section 6 for the experimental results.

We now discuss the algorithm complexity. For simplicity, we assume that each test sequence is of length $k$, i.e., it contains $k$ input symbols. Suppose that there are $N$ test sequences in the test set $S$. Since there are $N \log N$ comparisons to sort $S$ and each comparison takes time $O(k)$, it takes time $O(kN \log N)$ to sort all the tests in $S$. We then examine all the tests in the sorted order and remove all the prefixes; the total time is $O(kN)$. Since the total lengths of all the tests in $S$ is: $M=kN$, we have:

**Proposition 6.** It takes time $O(M \log N)$ to select tests using Algorithm 3 where $M$ and $N$ are the total length and number of all the test sequences in the given test set.

The cost of the test selection of Algorithm 3 is dominated by sorting the test sequences. We now discuss another method, using trees. It is optimal in terms of run time; it takes time proportional to the total lengths of the test sequences to be selected.

We grow a tree as follows. Initially, we have a root node $v_0$. Each edge is labeled with an input symbol. For each input sequence (test), we walk down the tree as follows. At a node $u$ of the tree with an input symbol $a$ in the sequence, if there is an outgoing edge from $u$ labeled with $a$, then we walk down along that edge, arrive at the end node $v$, and process the input symbol in the test sequence after $a$. However, if there are no outgoing edges labeled with $a$, then we add a tree edge $(u,v)$ from $u$, label it with $a$, walk down along $(u,v)$, and continue processing from $v$ as before. Each test requires a tree walk. After all the tests have been processed, we can obtain the selected tests as follows. We only have to select the tests (or paths) from the root to a leaf node; all the other tests (paths) a prefix of these selected tests. Note that each step of the tree walk takes a constant time (if we have a bitmap at each node to keep track of the labels of all the outgoing edges). Therefore, it takes time proportional to the length of a test sequence to process. Hence the total cost is $O(M)$.

**Proposition 7.** It takes time $O(M)$ to select tests using Algorithm 4 where $M$ is the total length of all the test sequences in the given test set.

Obviously, Algorithm 4 removes the prefixes and selects the same test set as Algorithm 3. It is more efficient but involves an on-line tree construction process. Again, if the given test set is a covering set then the selected subset
is also a covering. But it may contain redundant tests in terms of covering
the reachability graph, of which we have no information.

Algorithm 4.
input: test sequence set \( S \) and alphabet set of all the input symbols \( \Sigma \)
output: subset \( S^* \) of \( S \) with redundant tests removed

1. \( V = \{ v_0 \} \); /* \( V \) is node set of tree \( T \); each node keeps track of all
   input symbols of its outgoing edges by a bitmap, */
2. for \( i = 1, \ldots, N \) /* \( N \) is the number of test sequences in \( S \) */
3. \( u = v_0 \); /* a tree walk from initial node \( v_0 \) */
4. for \( j = 1, \ldots, k_i \) /* \( k_i \) is number of input symbols in sequence \( s_i \)
   */
5. let \( t_{ij} \) be transition of \( s_i \) under examination with input \( t_{ij}>a \);
6. if \( u . \text{bitmap}(a)=0 \) /* not recorded yet */
7. initialize a new node \( v \);
8. construct a new tree edge \( (u, v) \);
9. \( u . \text{bitmap}(a)=1 \); /* record this input symbol at \( u \) */
10. \( u = v \); /* walk down tree to \( v \) along newly constructed edge */
11. else /* input has been recorded and tree edge exits for a walk */
12. \( u = v \);
13. \( S^* = \Lambda \); /* an empty set */
14. for each leaf node \( v \) of tree \( T \);
15. let \( s \) be test sequence corresponding to path from \( v_0 \) to \( v \);
16. \( S^* = S^* \cup \{ s \} \);
17. return \( S^* \)

Figure 4. Algorithm 4

6. EXPERIMENTS

We applied our test selection algorithms to a simplified version of Q.931
protocol. We now report experimental results. Due to the space limit, for
the reachability graph of Q.931 and the test sequences, see [LH].

Q.931 is the connection control protocol for ISDN, comparable to TCP in
the Internet protocol stack. It manages connection setup and teardown.
However, it does not provide flow control nor perform retransmission, since
the underlying layers are assumed to be reliable and the circuit-oriented
nature of ISDN allocates bandwidth in fixed increments of 64 kbps. Similar
to TCP, Q.931 documents both the protocol itself and a protocol state
machine. For experiment purpose, we constructed an EFSM for Q.931 that covers most of the states and most of the message types. The corresponding reachability graph has 13 nodes and 19 transitions. We also have a set of 59 test sequences for Q.931 that were generated from an automated test generation tool.

We implemented Algorithm 1 and its three variations, and then applied to the 59 test sequence set with the reachability graph. We have:

Table 1. Comparison of Algorithm 1 and its variations

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Number of selected test sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>original test set</td>
<td>59</td>
</tr>
<tr>
<td>Algorithm 1</td>
<td>6</td>
</tr>
<tr>
<td>Algorithm 1.1 (sort-by-length)</td>
<td>9</td>
</tr>
<tr>
<td>Algorithm 1.2 (random-select)</td>
<td>8</td>
</tr>
<tr>
<td>Algorithm 1.3 (sort-by-alphabet)</td>
<td>6</td>
</tr>
</tbody>
</table>

From Table 1, all four algorithms reduce the number of final test sequences. According to the discussion in Section 4, Algorithms 1 and 1.3 tend to generate the least number of test sequences. Experimental results in Table 1 verify this observation; both algorithms have selected a set of 6 test sequences, which have the same edge coverage of the reachability graph as the original 59 sequences. Comparing with Algorithm 1 and 1.3, we observe that Algorithm 1.1 and 1.2 execute a little bit faster when processing the same set of input sequences. However, the number of test sequences selected by Algorithm 1.1 and 1.2 is larger than Algorithm 1 and 1.3 (9 and 8 compared to 6); efficiency is achieved with a cost of more redundancy in the tests selected.

We also compare Algorithm 3 and 4 with Algorithm 1 in the number of selected tests and run cost. They use the same set of test sequences as input. The following are the results:

Table 2. Comparison of Algorithm 3, 4 and 1

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Algorithm 3</th>
<th>Algorithm 4</th>
<th>Algorithm 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of selected tests</td>
<td>18</td>
<td>18</td>
<td>6</td>
</tr>
</tbody>
</table>

From Table 2, Algorithm 3 and 4 both have selected 18 sequences. The run time of Algorithm 4 is less than Algorithm 3. Comparing with Algorithm 1, both Algorithm 3 and 4 selected 12 more test sequences, since Algorithm 3 and 4 do not incorporate the information of the reachability graph of the
underlying system. On the other hand, the cost of Algorithm 3 and 4 is much less than that of Algorithm 1.

7. CONCLUSION

In this paper, we have discussed the test sequence selection problem. Given a test sequence set, we want to select a subset of tests; without sacrificing the coverage we want to minimize the number of the selected tests to reduce the test execution time. We have proposed several algorithms with different required information and coverage, at different costs, and with different redundancy of the selected tests.

Algorithm 1 and its variations Algorithm 1.1-1.3 all require that a reachability graph of the underlying protocol system is available. They are designed for Problem 1 where the coverage is the reachability graph. Algorithm 2 relaxes the requirement; it needs an EFSM of the underlying system. It is designed for Problem 2 where the coverage is also the reachability graph yet only the EFSM of the underlying system is given. Algorithm 3 and 4 are for the most general case when no information is needed for the test selection except for the given test set. They are designed for Problem 3 where there is no information available of the underlying system, and the selected tests also cover the reachability graph if the given test set does.

Algorithm 1 is expensive; it tends to minimize the number of tests, i.e., to reduce the redundancy as much as possible. Algorithm 2 has a same coverage as Algorithm 1, consumes less space (no reachability graph is needed), but takes more time to run. It shows the advantage of using the compact representation EFSM rather than its reachability graph. However, the on-line construction and recording of its edges involve more complicated data structures and algorithms, and it takes longer to run the algorithm. Algorithm 3 costs less than Algorithm 1 and 2. Algorithm 4 is optimal, since its cost is proportional to the total length of tests to be selected; one has to at least read all of them to select.

Algorithm 1 and 2 require more information and is more expensive to run. But they select a smaller test set without sacrificing the coverage. Algorithm 3 and 4 require no information of the underlying system and cost less, but select more tests. Of course they also cover the reachability graph if the given test set does. However, they may contain redundant tests, which are not needed for the coverage of the reachability graph. But there is no way
one can figure out since there is no information of the underlying protocol system at all.

In practice, often information of the underlying protocol system is not available or it is too expensive to obtain. In this case we resort to Algorithm 3 and 4. As demonstrated by the experiments, they can also reduce significantly the number of tests yet without sacrificing the coverage.

So far we assume that the underlying protocol system is deterministic, and in this case, a valid test sequence corresponds to a unique path in the reachability graph. If the system is non-deterministic, then a valid test sequence may correspond to multiple paths in the graph. Since the execution sequence is non-deterministic, the coverage of the edges in the reachability graph is probabilistic. While the union of all the possible paths, which are associated with a valid test sequence, is considered for coverage, we can only claim in a probabilistic sense.

ACKNOWLEDGEMENTS

Kun Du from Peking University implemented some of the algorithms. Gerard Holzmann and Margaret Smith from Bell Labs provided us with the set of Ubet scenarios used in our experiments. Our thanks go to the reviewers for their constructive comments and suggestions.

REFERENCES


EXECUTABLE TEST SEQUENCE FOR THE PROTOCOL DATA FLOW PROPERTY¹

Wen-Huei (Paul) Chen
Department of Electronic Engineering, Fu Jen Catholic University, Taipei, Taiwan, R.O.C.
paultaipei@yahoo.com.tw

Abstract
A new test sequence generation method is proposed for testing the conformance of a protocol implementation to its data portion modeled by an Extended Finite State Machine which is represented by a Data Flow Digraph. All-Use and IO-df-chain are two important criteria in selecting paths from the Data Flow Digraph in order to generate a test sequence which traces the data flow property, but it is a tedious process to select a path which satisfies the criteria while guaranteeing that the generated test sequence is executable (i.e., one that has feasible parameter values.) The proposed four-step method automatizes the selecting process as well as optimizing the test sequence length. First, the Data Flow Digraph is embedded with certain (but not all) parameter values so as to construct a Behavior Machine Digraph where executable test sequences can be directly generated. Second, executable test paths which trace every association defined by each criterion are generated from the Behavior Machine Digraph. Third, the Behavior Machine Digraph is embedded with these test paths so as to construct the SelectUse and SelectIO Digraphs. Finally, the Selecting Chinese Postman Tours of the two digraphs are used to generate the optimally executable test sequences that satisfy the All-Use and IO-df-chain criteria.

Keywords: Conformance testing, executable test sequence, data flow.

1. INTRODUCTION

A distributed system is composed of many parties (i.e., computers, instruments, etc.) remotely connected by communication links (i.e., cables,

¹ This work is supported by the National Science Council of Taiwan under Grant NSC892213E030025 and by Fu Jen University under a grant from the SVD section.
fibers, etc.) through which messages are transmitted, a protocol is the representation as well as the orderly exchange of these messages that must be agreed on by any party before using it, and a set of protocols is usually layered to establish a complex communicating behavior. In each party, a protocol is implemented in either software or hardware that has an upper (or lower) interface to the upper-layer (lower-layer) protocol(s) [9]. The objective of protocol conformance testing is to see if a protocol implementation conforms to the protocol specification defined as a standard. In a testing center, the protocol implementation is tested as a black box. Inputs are sent from an external tester to the implementation through the interfaces, and the outputs are checked to see if they are as expected. The sequence of input/output pairs is the test sequence, and the number of inputs is the test sequence length [5]. In general, such a test sequence is generated from the protocol specification.

The protocol specification contains a control and a data portion [2]. The control portion determines how messages are sent and received. It can be considered a deterministic² Finite State Machine (FSM) which contains states and transitions [5]. Initially, the FSM is in a specific state called the initial state. An input message (i.e., input or stimulus) will cause the FSM to generate output message(s) (i.e., outputs or responses) and to change from the current state to a new state; this process is a transition. The data portion specifies other functions (e.g., quality of service) that involve the parameter values associated with the messages. Informally, the data portion is described in words which are then formulated into a set of rules among parameter values [6]. Formally, the data portion is specified by an Extended Finite State Machine (EFSM) [2] which is extended from the FSM by introducing variables and parameters. Initially, the EFSM is in the initial state and all the variables are set to initial values. The EFSM can receive an input that has parameter values, which combine with certain variable values to define a logic function (i.e., predicate.) That input will cause the EFSM to change from the current state to a new state that depends on the truth value of the predicate, then the EFSM updates the variable values according to a computation function (i.e., action) while generating output(s) that have certain parameter values.

In a data portion described by a set of rules, the generated test sequence is executable if it has parameter values that do not violate any rule [1][6]. In [6], Kwast converts the FSM and rules into a Behavior Machine Digraph, in which paths can be used to generate the executable test sequence which verifies the rules. In [1], Chen converts the Behavior Machine Digraph into

² “deterministic” means that for each input there is at most one transition defined at each state.
a \textit{Selecting Digraph} and proposes the \textit{Selecting Chinese Postman Algorithm} to find a specific tour of the digraph which generates a minimum-length executable test sequence that verifies each rule at least once. In [2], Chen generalizes the Selecting Digraph into a \textit{SelectO} digraph which is used to generate an executable test sequence that verifies both the control and data portions. However, the data portions of many protocols which are described in words cannot be formulated into rules. This paper is concerned with test sequence generation from the data portion specified by an EFSM.

The EFSM of a data portion can be represented by a Data Flow Digraph, where the inputs, outputs, predicates and actions of the EFSM are represented by a set of nodes [11]. A test sequence generated from a path of the digraph is \textit{executable} (or \textit{feasible}) if it has certain input parameter values which make each predicate along the path to remain true. Because it is infeasible to traverse all possible paths of the Data Flow Digraph, a test path is usually selected according to a criterion which involves a data flow property of the EFSM. In [12], a criterion is defined for observing the data flow abnormality of the EFSM due to a fault model. However, it is difficult for a test designer to construct a fault model that covers all possible faulty implementations of the EFSM. In [10][11], the \textit{All-Use} and \textit{IO-df-chain} criteria are defined for observing the data flow of variable values from where they are defined (or input) to where they are used (output.) They have become important criteria in EFSM testing.

A test path of the Data Flow Digraph is said to satisfy the All-Use (or IO-df-chain) criterion if it can trace all the variable value associations defined by that criterion. In order to trace\footnote{such a trace is different from the “trace” used in verification for proving the property.} an association, the test path starts from the first node (i.e., the initial state) and proceeds to a specific node where a property of that association is exhibited, then returns to the first node to result in a \textit{complete path} [11]. At the same time, we must ensure that the complete path allows a feasible assignment of input parameter values which makes all predicates along the path remain true. New complete paths must be obtained until that all associations defined by the criterion are traced. Unfortunately, this process is tedious and introduces a lot of overhead sequences in taking the EFSM to/from the initial state. In this paper, we are going to propose an automatic method to generate the executable test sequences that satisfy the All-Use and IO-df-chain criteria as well as optimizing the test sequence length.

We convert the Data Flow Digraph (the EFSM) into a \textit{Behavior Machine Digraph}, the paths of which can be used to generate executable test sequences. Unlike the \textit{Global FSM} which represents the complete behavior of the EFSM by enumerating all possible parameter and variable values as the inputs/outputs and states, the Behavior Machine only represents a partial
behavior which will be used for testing. That is, only certain parameter (or variable) values are embedded into the inputs/outputs (states) for constructing the Behavior Machine where all predicates are removed [4]. For instance, the Simple Connection Protocol of 4 states (see next section) is converted to a Global FSM of 768 states but is converted to a Behavior Machine of 6 states. Multiple paths of the Behavior Machine Digraph trace the same variable value association defined by the criterion, but only one needs to be included into the final test sequence. Thus, we use the Selecting Chinese Postman Algorithm to optimally select either one to construct a minimum-length executable test sequence which traces each association at least once.

In Section 2, the two criteria are reviewed. In Section 3, the method based on the first criterion is proposed. In Section 4, the method based on the second criterion is proposed. In Section 5, our conclusions are presented.

2. THE TWO DATA FLOW TESTING CRITERIA

Consider the Simple Connection Protocol (SCN). The control portion specifies how to establish/release a connection. To establish a connection, an input “CONreq” (i.e., connection request) from the upper interface causes the SCN to output “connect” to the lower interface. Inputs “accept” or “refuse” from the lower interface causes the SCN to output “CONrsp(+)” (i.e., positive connection response) or “CONrsp(-)” (negative connection response) to the upper interface. After the connection is established, an input “Data” from the upper interface causes the SCN to output “data” to the lower interface. To release the connection, an input “Reset” from the upper interface cause the SCN to output “abort” to the lower interface. Notice that the uppercase (or lowercase) first letter indicates that the message is related to the upper (lower) interface.

The data portion specifies other functions of the SCN by two types of variables. Variables of the first type are called the memory variable that will store temporary values. In the SCN, the memory variable TryCount stores the number of unsuccessful connection attempts and has values from 0 to 2, and the memory variables ReqQos and FinQos store the levels of requested and final quality of service and have values from 0 to 3. Variables of the second type are called parameter variables that will store the parameter values of inputs. Let X(y) denote an input X which has a parameter y, then the parameter variable denoted by X.y will store the value of parameter y. This definition can be extended to multiple parameters. In the SCN, the parameter variables CONreq.qos, accept.qos and data.qos will
store the values of the parameter “qos” of inputs “CONreq(qos)”, “accept(qos)” and “data(qos)” respectively. They have values from 0 to 3.

The EFSM can be represented by the Data Flow Digraph of Figure 1,

*Figure 1. The Data Flow Digraph of the SCN protocol*

where the states, inputs, outputs and actions are represented by the state nodes, input nodes, output nodes and action nodes which enclose symbols that start with “s”, “i”, “o” and “a” respectively. Each predicate of the EFSM is represented by one predicate-decision node (i.e., decision node) and two predicate-outcome nodes (i.e., predicate nodes) which specify the two possible outcomes. The decision and predicate nodes enclose symbols that start with “d” and “p” respectively. The execution sequence of nodes is indicated by edge directions. For example, consider Figure 1. In the input node “i1”, the value of variable CONreq.qos will be input according to the statement “Input CONreq(qos).” Then, in decision node “d1”, the value of CONreq.qos is decided to see whether it is larger than 1 or not. Then, the EFSM may either arrive at either the predicate node p1 or predicate node p2.

---

4 In [10][11], a predicate is represented by a predicate node and the results of the predicates are represented by edges leaving that node.
which represents the two possible outcomes “CONreq.qos > 1” and “CONreq.qos ≤ 1.”

In Figure 1, a path is a sequence of nodes that are connected by edges. A tour is a special path which starts and ends at the same node. A path can be used to generate a test sequence by considering only the inputs and outputs in the path. For example, the path \([s_1, a_1, a_2, a_3, s_2, i_1, a_4, d_1, p_1, o_1, s_2]\) is used to generate the test sequence \([\text{Input CONreq(qos)}, \text{Output Nonsupport(qos)}]\) (or simply \([\text{CONreq(qos)}/\text{Nonsupport(qos)}]\)) by considering only input “i1” and output “o1.” In reality, the parameter “qos” must be given a value which makes predicate \(p_1\) hold true. For example, \([\text{CONreq(0)}/\text{Nonsupport(0)}]\) is not executable because predicate \(p_1\) will not hold true, but \([\text{CONreq(2)}/\text{Nonsupport(2)}]\) is executable. An executable path is a path that allows parameter values to be assigned which causes each predicate in the path to be true in order to generate the executable test sequence. Not all paths are executable. For example, the path \([s_1, a_1, a_2, a_3, s_2, i_1, a_4, d_1, p_2, o_2, s_3, i_2, d_2, p_4, o_3, s_1]\) is not executable. It is because that the value of variable \(\text{TryCount}\) becomes “1” in node \(a_1\). In order to pass through predicate node \(p_4\) which requires that the value of variable \(\text{TryCount}\) be equal to 2, the variable should increase its value in the intermediate nodes. However, the only node that can increment the value of \(\text{TryCount}\) is node \(a_5\), and that node is not in the path.

All-Use and IO-df-chain are two important criteria in selecting a test path from the Data Flow Digraph [11]. The first criterion claims that we should trace each variable from where it is defined (i.e., its value is first assigned) to where that value is used. We first describe where and when a variable is defined. At an input node, a variable is defined by an input statement. For example, at input node \(i_1\) of Figure 1, variable \(\text{CONreq.qos}\) is defined by the input statement “Input CONreq(qos)” which gives variable \(\text{CONreq.qos}\) a value. At an action node, a variable is defined by a computation statement which gives the variable a value. For example, at action node \(a_1\), variable \(\text{TryCount}\) is defined by the statement “\(\text{TryCount} := 0\)” where variable \(\text{TryCount}\) is assigned the value of 0. We then describe where and when a variable is used. At a predicate node, a variable is used in the predicate statement where the value of the variable will determine the result of the predicate. For example, in Figure 1, variable \(\text{CONreq.qos}\) is used at predicate node \(p_1\) because the value of \(\text{CONreq.qos}\) will determine whether the predicate “\(\text{CONreq.qos} > 1\)” is true or not. At an action node, a variable is used in the computation statement where the variable value will determine the value of another variable. For example, at action node \(a_6\), variable \(\text{ReqQos}\) is used in the statement “\(\text{FinQos} := \min(\text{accept.qos}, \text{ReqQos})\)” where the value of \(\text{ReqQos}\) determines the value of variable \(\text{FinQos}\). At an output node, a variable is used in the output statement where the value of the variable is output. For example, at node \(o_2\), variable \(\text{ReqQos}\) is used in the
output statement “Output connect(ReqQos).” Table 1 lists the variables defined and used in Figure 1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Node where it is defined</th>
<th>Node where it is used</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONreq.qos</td>
<td>i1</td>
<td>a4 p1 p2</td>
</tr>
<tr>
<td>accept.qos</td>
<td>i3</td>
<td>a6</td>
</tr>
<tr>
<td>TryCount</td>
<td>a1 a5</td>
<td>a5 p3 p4</td>
</tr>
<tr>
<td>ReqQos</td>
<td>a2 a4</td>
<td>a6 o2 o1</td>
</tr>
<tr>
<td>FinQos</td>
<td>a3 a6</td>
<td>o5 o6</td>
</tr>
</tbody>
</table>

Table 1. Variables defined and used at the nodes of the Data Flow Digraph of Figure 1

In tracing a variable X from node J (where X is defined) to node K (where X is used), the All-use criterion claims that the variable cannot not be redefined in the tracing process. Thus, to trace such an association, we must construct the define-clear-use path\(^5\) which connects node J to node K in such a way that the first node is the only node of the path where X is defined. If such a define-clear-path exists, we say that variable X and nodes J and K form an association called a define-use pair (or du-pair) du(J, K, X). For example, consider Figure 1 and Table 1. Variable ReqQos and nodes a4 and o2 form a du-pair du(a4, o2, ReqQos) because it can be traced by the define-clear-use path [a4, d1, p2, o2], where variable ReqQos is defined exclusively at node a4 and used at node o2. It is possible that many define-clear-use paths can trace the same du-pair. But such a path may not exist at all so that not all associations of variables and nodes form du-pairs. A path (or test sequence) of the Data Flow Digraph is said to satisfy the All-Use criterion if all possible du-pairs can be traced.

The second criterion (i.e., the IO-df-chain criterion) claims that we should trace the data flow from an input node J (where a variable X is defined) to an output node K (where a variable Y is used), where X and Y can be either the same or different variables. Variables X and Y are input and output parameter values that are controlled and observed by the testing system respectively. Certainly, the value of variable X may not directly affect the value of variable Y, but it may affect indirectly. That is, X may affect another variable that in turn affects variable Y, and so on. For example, consider the path [i1, a4, d1, p2, o2] of Figure 1. At node i1, the EFSM receive an input “CONreq” which has a parameter “qos,” the value of which is stored by the parameter variable CONreq.qos. At node a4, the value of CONreq.qos affects the value of variable ReqQos. At node o2, the value of ReqQos affects the parameter value of output “connect.” As a result, we can control the parameter value of the input “CONreq” to observe

---

\(^5\) the definition is extended from the define-clear path of [10][11].
the parameter value of the output “connect.” Obviously, a sequence of define-clear-use paths must be connected to trace this data flow.

By definition, a sequence of define-clear-use paths \( p_1, p_2, \ldots, p_{r-1}, p_r, \ldots, p_s \) (for variables \( X_1, X_2, \ldots, X_{r-1}, X_r, \ldots, X_s \) respectively) which starts from an input node \( J \) and ends at an output node \( K \) form an input-output chain (i.e., io-chain) \( \text{io}(J, X_1, K, X_s) \). The chain \( [p_1, p_2, \ldots, p_{r-1}, p_r, \ldots, p_s] \) can be used to trace the data flow from an input node \( J \) where variable \( X_1 \) is input to an output node \( K \) where the value of variable \( X_s \) is output, because the value of variable \( X_r \) is determined by the value of variable \( X_{r-1} \) through a computation statement in the node where \( p_{r-1} \) and \( p_r \) intersect, i.e., variable \( X_{r-1} \) affects \( X_r \) at the intersecting node (with the restriction that an action can have only one statement so that \( X_{r-1} \) cannot affect other variables). For example, \([i_1, a_4]\) is a define-clear-use path for variable \( \text{CONreq.qos} \) which starts from an input node, and \([a_4, d_1, p_2, o_2]\) is a define-clear-use path for variable \( \text{ReqQos} \) which ends at an output node. The two paths are connected into an io-chain \( \text{io}(i_1, \text{CONreq.qos}; o_2, \text{ReqQos}) \) which is \([i_1, a_4, d_1, p_2, o_2]\). A path (or test sequence) of the Data Flow Digraph is said to satisfy the IO-df-chain criterion if all possible io-chains are traced.

3. THE METHOD FOR THE FIRST CRITERION

In Section 2, we have described the All-Use and IO-df-chain criteria that are guidelines for selecting the test paths from the Data Flow Digraph. In this section, we will propose a method which automatically generates the executable test sequence that satisfies the first criterion. Our method involves four steps.

The first step involves the conversion of the Data Flow Digraph (Figure 1) into the Behavior Machine Digraph (Figure 2) where executable paths can be directly generated. In general, a node \( J \) (Figure 1) is converted into nodes \( J_1, J_2, J_3, \ldots \) (Figure 2)\(^6\) by embedding parameter values. If all parameter values are embedded, the Behavior Machine Digraph will be very large. Hence, we only embed parameter values that enable the removal of decision nodes. For example, consider node \( i_1 \) (i.e., \( \text{CONreq(qos)} \)) of Figure 1. The parameter \( \text{qos} \) has values from 0 to 3, so that we can convert node “\( \text{CONreq(qos)} \)” into four nodes “\( \text{CONreq(0)} \)”, “\( \text{CONreq(1)} \)”, “\( \text{CONreq(2)} \)”, “\( \text{CONreq(3)} \)” . However, because either “\( \text{CONreq(0)} \)” or “\( \text{CONreq(1)} \)” can make predicate \( p_2 \) true (and either “\( \text{CONreq(2)} \)” or “\( \text{CONreq(3)} \)” can make predicate \( p_1 \) true), we only create node “\( \text{CONreq(1)} \)” (i.e., node \( i_1 1 \) ) and node

\(^6\) Without loss of generality, notations \( J_1, J_2, J_3, \ldots \) of the Behavior Machine Digraph represent the nodes converted from the node \( J \) of the Data Flow Digraph in this paper.
“CONreq(2)” (i.e., node i1\textsubscript{2}). Then we remove the decision node d1 from the digraph since no decisions are required there. Because every predicate in the digraph is true, any path of the digraph can be used to generate an executable test sequence. In Figure 2, the Behavior Machine is represented by fine lines and text, but it includes some bold characters that will be used in the next step for tracing the data flow. In general, a bold character “\textit{X_{def}}” (or \textit{X_use}) is put on nodes J\textsubscript{1}, J\textsubscript{2}, J\textsubscript{3}, .. of Figure 2 if variable \textit{X} is defined (used) at node \textit{J} of Figure 1. For example, a bold character \textit{C_{def}} (an abbreviation of CONreq.qos\textsubscript{def}) is put on nodes i1\textsubscript{1} and i1\textsubscript{2} of Figure 2, because variable CONreq.qos is defined at node i1 of Figure 1 (see Table 1.)

\begin{itemize}
  \item \textit{C}: CONreq.qos
  \item \textit{A}: accept.qos
  \item \textit{T}:TryCount
  \item \textit{R}: ReqQos
  \item \textit{F}:FinQos
\end{itemize}

Remark: \textit{C_{def}} (\textit{use}): variable defined (used) d at that node

\textit{Figure 2.} The Behavior Machine Digraph of the Data Flow Digraph of Figure 1 (only new statements other than that of Figure 1 are indicated)
The second step involves finding executable test paths of Figure 2 that can trace the du-pairs of Table 1. Consider the du-pair du(J, K, X), where variable X is defined at node J and used at node K. As described in Section 2, this du-pair is traced by a define-clear-use path of Figure 1 which starts from node J and ends at node K and does not redefine the variable in the intermediate nodes. In Figure 2, such an executable define-clear-use path corresponds to specific paths which start from nodes J, J2, ..., Jr and end at nodes K1, K2, ..., Kq where the symbols “X_def” can only be seen in the starting nodes. These specific paths of Figure 2 can be constructed from shortest paths of Figure 2 where nodes which have the symbol X_def (except the starting nodes) and their adjacent edges are (temporarily) removed. For example, to trace the du-pair du(i1, a4, CONreq.qos) of Table 1 where “CONreq” is defined at node i1 and used at node a4, we find the specific shortest paths which starts at nodes i1 and i1 and ends at nodes a4 and a4 of Figure 2. These shortest paths are used to produce the executable define-clear-use paths [i1, a41] and [i1, a42] that can trace the du-pair du(i1, a4, CONreq.qos). The complete executable define-clear-use paths of Figure 2 for tracing the du-pairs of Table 1 are shown in Table 2.

<table>
<thead>
<tr>
<th>Label</th>
<th>du-pairs2 of Table 1</th>
<th>Executable define-clear-use paths of Figure 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>du(i1,a4, CONreq.qos)</td>
<td>[i11, a41] or [i12, a42]</td>
</tr>
<tr>
<td>2</td>
<td>du(i1,p1, CONreq.qos)</td>
<td>[i11, a41, p1]</td>
</tr>
<tr>
<td>3</td>
<td>du(i1,p2, CONreq.qos)</td>
<td>[i12, a42, p2]</td>
</tr>
<tr>
<td>4</td>
<td>du(i3, a6, accept.qos)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>du(a1, a5, TryCount)</td>
<td>[a1, a2, a3, s2, i12, a42, p2, o21, s31, i21, p31, a51]</td>
</tr>
<tr>
<td>6</td>
<td>du(a, p3, TryCount)</td>
<td>[a1, a2, a3, s2, i12, a42, p2, o21, s31, i21, p31]</td>
</tr>
<tr>
<td>7</td>
<td>du(a5, p3, TryCount)</td>
<td>[a51, o22, s32, i22, p32]</td>
</tr>
<tr>
<td>8</td>
<td>du(a5, p4, TryCount)</td>
<td>[a52, o23, s33, i23, p4]</td>
</tr>
<tr>
<td>9</td>
<td>du(a4, o2, ReqQos)</td>
<td>[a42, p2, o21] or [2, p2, o21, s31, i21, p31, a51, o22] or [a42, p2, o21, s31, i21, p31, a51, o22, s32, i22, 2, a52, o23]</td>
</tr>
<tr>
<td>10</td>
<td>du(a4, a6, ReqQos)</td>
<td>2, p2, o21, s31, i3, a6</td>
</tr>
<tr>
<td>11</td>
<td>du(a4, o1, ReqQos)</td>
<td>[a41, p1, o1]</td>
</tr>
<tr>
<td>12</td>
<td>du(a6, o5, FinQos)</td>
<td>[a6, o5]</td>
</tr>
<tr>
<td>13</td>
<td>du(a6, o6, FinQos)</td>
<td>[a6, o5, s4, i5, o6]</td>
</tr>
</tbody>
</table>
The third phase involves constructing a SelectUse Digraph of Figure 3 by embedding the executable test paths of Table 2 into the Behavior Machine Digraph of Figure 2. Generally, an executable define-clear-use path from node J to node K is embedded as a bold edge from node J to node K, and a bold label is put on the path to indicate the du-pair traced by the path. Cost is assigned to the edge according to the number of inputs that it contribute to the final test sequence when that edge is traversed in the test path, because the length of a test sequence is decided by the number of inputs that it has. Thus, a fine edge is assigned a cost of 1 if it leaves an input node, and a bold edge is assigned a cost which is the number of inputs contained in the define-clear-use path represented by the bold edge. For example, consider the executable test path \([a1, a2, a3, s2, i1, a4, p2, o2, s3, i2, p3, a5]\) of Table 2 that traces the du-pair du(a1, a5, TryCount), which has the label “5.” Because the path starts at node a1 and ends at node a5, it is embedded as a bold edge from node a1 to a5. The bold edge is labeled with “5” and assigned a cost of 2. An executable define-clear-use path that can trace multiple du-pairs is embedded as an edge that has multiple labels. For example, the define-clear-use path described above can trace two du-pairs of Table 2, and we put labels “5” and “6” on it. For many bold edges that share the same label, only one edge needs to be included in the final test path.
because these bold edges represent the executable test paths that trace the same du-pair. As a result, the Selecting Chinese Postman Tour of the SelectUse Digraph (which is a minimum-cost tour where each type of label appears at least once) can be used to generate a minimum-length executable test sequence that traces all the du-pairs.

The fourth step involves using the Selecting Chinese Postman Algorithm proposed in [1] to find the Selecting Chinese Postman Tour of the SelectUse Digraph in order to generate the executable test sequence. The algorithm contains two phases. The first phase replicates (or deletes) each edge of the SelectUse digraph, resulting in a minimum-cost Selecting Symmetric Augmentation (MCSSA) which satisfies the properties that i) each node has the same number of entering and leaving edges and ii) each type of label appears in the digraph at least once. The MCSSA can be obtained by solving a system of integer programming equations formulated from the SelectUse Digraph. The integer programming problem is solved by a branch and bound algorithm which iterates to improve an initial solution [8]. When the problem size is not very large, the branch and bound algorithm can find the optimal solution. Otherwise, the branch and bound algorithm will at least obtain a significantly improved solution if we can make the algorithm run at long time. In the second phase, we check whether the MCSSA is an Euler Digraph or a collection of disjoint Euler Digraphs [7]. In the former case, an Euler Tour of the Euler Digraph is an Selecting Chinese Postman Tour of the SelectIO Digraph. In the latter case, a tour will be used to connect these disconnected Euler Digraphs into an Euler Digraph, so that the Euler Tour algorithm can be applied. For Figure 3, the Lindo package [8] has solved the integer programming equations in less than one second and results in an Euler Digraph, resulting in the first case so that the optimal tour is found. The Selecting Chinese Postman Tour of Figure 3: \[[s1, a1, a2, a3, s2, i1, 1, a4_1, 11, o_1, s2, i1_1, 2, p_1, o_1, s2, i1_2, 1, a4_2, 10, a6, (13, 12), o_6, s4, i4, o_4, s1, a1, a2, a3, s2, i1_2, 3, p_2, o_2_1, s3_1, i3, 4, a6, 12, o_5, s4, i4, o_4, s1, a1, a2, a3, s2, i1_2, 1, a4_2, 9, o_2_2, s3_2, i2_2, p_3_2, a5_2, o_2_3, s3_3, i2_3, p_4, o_3, s1, a1, (6, 5), a5_1, 7, p_3_2, a5_2, 8, p_4, o_3, s1, a1, a2, a3, s2, i1_2, 3, p_2, o_2_1, s3_1, i3, 4, a6, 12, o_5, s4, i4, o_4, s1]\] is used to generate the minimum-length executable test sequence \[[i_1_1, o_1, i_1_1, o_1, i_1_2, o_2_1, i_3, o_5, i_5, o_6, i_4, o_4, i_1_2, o_2_1, i_3, o_5, i_4, o_4, i_1_2, o_2_1, i_2_1, o_2_2, i_2_2, o_2_3, i_2_3, o_3, i_1_2, o_2_1, i_2_1, o_2_2, i_2_2, o_2_3, i_2_3, o_3, i_1_2, o_2_1, i_3, o_5, i_4, o_5]\] which traces all the du-pairs of Table 1.

4. THE METHOD FOR THE SECOND CRITERION

In Section 3, we have proposed a method to produce an executable test sequence which satisfies the first criterion by finding the Selecting Chinese
Postman Tour of the SelectUse Digraph. In this Section, we are going to extend the method to generate an executable test sequence which satisfies the second criterion, i.e., the IO-df-chain criterion which requires tracing each io-chain at least once. The extended method contains two steps.

The first step involves finding all io-chains of Figure 1. In the SelectUse Digraph of Figure 3, we consider input nodes $J_1, J_2, J_3, ..., \text{(converted from node J where variable X is defined)}$ to output nodes $K_1, K_2, K_3, ... \text{(converted from node K where variable Y is used).}$ As described in Section 2, an io-chain is constructed from a sequence of define-clear-use paths so as to connect an input node to an output node (but notice that a sequence of define-clear paths defined in [11] does not necessarily form an input-output-chain). We call those bold edges which leave input nodes $J_1, J_2, J_3, ..., \text{that are the define-clear-use paths for variable X as the (X-J) edges,}$ and those bold edges which enter output nodes $K_1, K_2, K_3, ... \text{that are the define-clear-use paths for variable Y as the (Y-K) edges.}$ A path which involves only bold edges is a bold path. In the SelectUse Digraph, the shortest bold path from these (X-J) edges to those (Y-K) edges can be used to generate the sequence of define-clear-use paths which can trace io-chain $\text{io(J, X, K, Y).}$ Those specific shortest paths of the SelectUse Digraph can be obtained from shortest paths of the SelectUse Digraph where fine edges are (temporarily) removed. For example, the bold edge (i.e., the define-clear-use paths $[i1 1, I, a4_1])$ and another bold edge (i.e., $[a4_1, 11, o1])$ compose an io-chain $\text{io(i1,CONreq.qos, o1, ReqQos) = [i1 1, I, a4_1, 11, o1].}$ The complete io-chains for Figure 1 obtained from these specific shortest paths of Figure 3 are shown in Table 3, where each io-chain is given a label.

### Table 3. The io-chains for the Data Flow Digraph of Figure 1

<table>
<thead>
<tr>
<th>Label</th>
<th>input-output-chains (Figure 1)</th>
<th>Composed define-clear-use paths (Figure 3)</th>
<th>Corresponding Executable Paths (Figure 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong></td>
<td>$\text{io(i1,CONreq.qos, o1, ReqQos)}$</td>
<td>$[i1 1, I, a4_1, 11, o1]$</td>
<td>$[i1 1, a4_1, p1 o1]$</td>
</tr>
<tr>
<td><strong>b</strong></td>
<td>$\text{io(i1,CONreq.qos, o2, ReqQos)}$</td>
<td>$[i1 2, I, a4_2, 9, o2 1]$</td>
<td>$[i1 2, a4_2, p2, o2 1]$</td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>$\text{io(i1, CONreq.qos, o5, FinQos)}$</td>
<td>$[i1 2, I, a4_2, 10, a6, 12, 05]$</td>
<td>$[i1 2, a4_2, p2, o2 1, s3 1, i3, a6, o5]$</td>
</tr>
<tr>
<td><strong>d</strong></td>
<td>$\text{io(i1, CONreq.qos, o6, FinQos)}$</td>
<td>$[i1 2, I, a4_2, 10, a6, 13, 06]$</td>
<td>$[i1 2, a4_2, p2, o2 1, s3 1, i3, a6, 05, s4, i5, o6]$</td>
</tr>
</tbody>
</table>
The second step involves constructing a SelectIO Digraph of Figure 4 by embedding the executable test paths of Table 3 to the Behavior Machine Digraph of Figure 2. Generally, an executable test path listed in Table 3 which starts from an input node J and ends at output node K is embedded as a bold edge from vertex J to vertex K. The label which represents the io-chain traced by the executable path is put on the edge (see Table 3 and Figure 4.) Costs are assigned to the edges of the SelectIO Digraph similar to the process described in the third step of Section 3 for assigning the cost to the edges of the SelectUse Digraph. A Selecting Chinese Postman Tour of the SelectIO digraph can be used to generate an minimum-cost executable test sequence that checks each io-chain. The tour is obtained using the algorithm described in the four step of Section 3. The Selecting Chinese Postman Tour of Figure 4: \([s1, a1, a2, a3, s2, i1_1, a, o1, s2, i1_2, b, o2_1, s3_1, i2_1, p3_1, a5_1, o2_2, s3_2, i2_2, p3_2, a5_2, o2_3, s3_3, i2_3, p4, o3, s1, a1, a2, a3, s2, i1_2, (c, d), o6, s4, i4, s1]\) is used to generate the executable test sequence \([i1_1, o1, i1_2, o2_1, i2_1, o2_2, i2_2, o2_3, i2_3, o3, i1_2, o2_1, i3, o5, i5, o6, i4, o4]\) which traces all the io-chains of Table 3.

*Figure 4.* The SelectIO Digraph constructed from the Behavior Machine Digraph of Figure 2 by embedding the io-chains of Table 3 as bold edges.
5. CONCLUSIONS

In this paper, we have proposed a method to automatically generate the executable test sequence from the protocol specification for verifying two data flow criteria, based on finding the Selecting Chinese Postman Tour of the SelectUse and SelectIO Digraphs constructed from the Data Flow Digraph of the protocol. The first phase of the Selecting Chinese Postman Algorithm involves solving a system of integer programming equations so as to find an augmentation. In our experiences of solving such equations [1][2], a linear programming version of the formulation always yield integer results. We want to check whether these equations satisfy a specific property so that the linear programming approach can be applied.

Our method of minimizing the test sequence length can be easily extended to minimizing the test sequence cost, by assigning cost to the edges of the digraph. And the method can be combined with the duplexE digraph method to generate a synchronizable and executable test sequence [3].

REFERENCES

A METHOD TO GENERATE
CONFORMANCE TEST SEQUENCES
FOR FSM WITH TIMER SYSTEM CALL

Takanori Mori†, Kohei Tokuda†,
Harumasa Tada†, Masahiro Higuchi‡ and Teruo Higashino†
†Dept. of Info. and Math. Sci., Osaka Univ., Toyonaka, Osaka 560-8531, Japan
{t-mori, k-tokuda, tada, higashino}@ics.es.osaka-u.ac.jp
‡School of Sci. and Eng., Kinki Univ., Higashiosaka, Osaka 577-8502, Japan
higuchi@ee.kindai.ac.jp

Abstract In this paper, we propose a method to generate conformance test sequences for communication protocols modeled as FSM with timers. The test sequences generated by the proposed method can detect any single fault of timer commands or destination states in the transitions on protocol machines. For each single timer command fault, we give sufficient conditions that a given IUT is (or is not) equivalent to its specification. Based on these sufficient conditions, we give a method for generating test sequences. For each destination state fault, we give a test sequence generation method based on Wp-method. To show the usefulness of this method, we have developed a system for generating test sequences, and applied it to DHCP (Dynamic Host Configuration Protocol). As a result, we have generated the test sequences efficiently.

Keywords: conformance testing, test case generation, embedded system, FSM, timer

1. INTRODUCTION

Recently, complex systems consisting of two or more cooperating components are commonly used. In such complex systems, interactions between components cannot be observed and controlled from their environment, usually[1]. Conformance testing for complex systems are discussed in a lot of papers[2, 3, 4].

Communication protocols which use timer functions provided by operating systems (OS) also can be viewed as such complex systems. In this paper, we discuss conformance testing of such protocols. A system considering in this paper consists of a FSM based protocol machine and timers. A protocol machine is modeled as a deterministic finite state machine (DFSM) which uses timer functions of OS through system
calls. The interactions between a protocol machine and timers cannot be observed and controlled from their environment.

In conformance testing for specific subsystems of complex systems, a given complex system is divided into two components; (i) $Spec$: the subsystem of the test target, and (ii) $Context$: the subsystems in the complex system other than $Spec$. The context is assumed to be correctly implemented. A system consisting of IUT (Implementation Under Test) and Context is called SUT (System Under Test). Such testing is called embedded testing[5] or gray box testing[2].

For embedded testing, we must consider the following features of complex systems; (i) although the IUT does not conform to its specification $Spec$, the behavior of the resulting SUT may conform to $Spec\cdot Context$, and (ii) a single fault of IUT may cause multiple faults of SUT.

Embedded testing has been mainly investigated by assuming that each component of SUT is modeled as DFSM[2, 3, 4]. Although timers can be modeled as DFSM, the numbers of states are usually large. We will consider the above features of timers.

The remainder of this paper is organized as follows. In Section 2, we describe the formal model of FSM with timers. In Sections 3 and 4, we discuss a test sequence generating method. In Section 5, we explain an experiment of applying the proposed method to DHCP.

2. COMMUNICATION PROTOCOLS WITH TIMER SYSTEM CALL

![Figure 1. Communication protocols with timer system call](image)

In this paper, we consider a class of communication protocols shown in Figure 1. In this model, a communication entity consists of a finite state protocol machine and multiple timers. The interactions between the protocol machine and timers are (i) timer commands (enabling or disabling timers) from the protocol machine to timers, and (ii) timeout notifications from timers to the protocol machine. Assume that the timer function is provided by operating systems. The timer commands are given as system calls, e.g. `set-timer` or `del_timer` in Linux. The
timeout notifications are given as signals. These interactions cannot be observed and controlled from their environment.

2.1. Timer

In general, operating systems can manage multiple timers. A user process can enable or disable individual timer through system calls. Here, the timer period from enabling a timer to expiring the timer is called the timer expiring time. The timer expiring time is specified as a parameter of the system call. After enabling a timer, if the timer expiring time elapses without disabling or re-enabling the timer, then the timer expires and produces a timeout signal.

We assume that each timer can be identified by a timer number and the timer expiring time is fixed for individual timer. A specification of timers is defined by a vector $T$ of timer expiring times. $v[i]$ denotes the $i$-th element of a vector $v$. $T[i]$ denotes the timer expiring time of timer $i$. In order to describe a state of timers, we introduce a timer value vector $\tau$. $\tau[i]$ denotes the current value of timer $i$. $\tau[i]$ has an integer value ($0 \leq \tau[i] \leq T[i]$) or $\bot$. $\tau[i] = \bot$ means timer $i$ is not active. We assume that $\bot > x$ and $\bot - x = \bot$, for all $x \in N$ ($N$: the set of non-negative integers). When timer $i$ is enabled, $\tau[i]$ is set to $T[i]$. When timer $i$ is disabled, $\tau[i]$ is set to $\bot$. The values of all $\tau[i]$ decrease one by one simultaneously every time unit. When $\tau[i]$ becomes zero, timer $i$ produces a timeout signal and $\tau[i]$ becomes $\bot$. We define $I[i]$ as the set $\{0, 1, \ldots, T[i], \bot\}$. $T$ denotes the set of timer value vectors $\{\tau | \tau[i] \in I[i]\}$.

If there exists a timer whose value is zero, a unit time never elapses. If there exist two or more timers whose values are zero, the timers produce timeout signals in the increasing order of their timer numbers. If a timer $i$ whose value is zero is re-enabled or disabled, $\tau[i]$ is set to $T[i]$ or $\bot$, respectively. Such a timer does not produce a timeout signal.

2.2. Protocol machine

A protocol machine is modeled as a Mealy deterministic finite state machine (DFSM) and defined as the following 6-tuple $(Q, X, n, Y, H, s_0)$.

$Q$ : a finite set of states.
$X$ : a finite set of external input symbols.
n$ : the number of timers.
$Y$ : a finite set of external output symbols.
$H$ : a finite set of transitions $(u, v, x, y, \bar{p})$.
$u, v \in Q$ : source state, destination state.
x $\in (X \cup \{1, \ldots, n\})$ : input.
y $\in Y$ : external output.
$\bar{p} \in \langle S, D, N \rangle^n$ : timer command vector.
s_0 : an initial state.
A state transition is executed when a protocol machine receives an input (an external input or a timeout signal). State transitions caused by external inputs and timeout signals are called external input transitions and timeout transitions, respectively. As protocol machines are deterministic, the current state and an input can determine the destination state and outputs (an external output and a timer command vector) uniquely. We assume that state transitions are executed instantaneously. A timer command vector is a vector \( \vec{p} = (p[1], p[2], \ldots, p[n]) \). \( p[i] \) denotes a command for timer \( i \). The command is either \( S \), \( D \), or \( N \). \( S \) is a command enabling a timer. \( D \) is a command disabling a timer. \( N \) denotes null operation for a timer.

We assume that protocol machines are reduced. We also assume that each protocol machine has a reliable reset feature. If a protocol machine receives a reset input, the protocol machine is reset to its initial state and disables all timers. If a protocol machine receives an undefined input, the protocol machine is reset and produces an error output. With such error transitions, protocol machines are completely specified.

**Figure 2.** Protocol re-transmitting a message when timeout occurs

![Diagram](image)

**Example 1** Figure 2 is a protocol machine which re-transmits messages when a timer expires. Transmitting a message to a receiver, the protocol machine enables both Timer 1 and Timer 2. The protocol machine re-transmits the message every expiration of Timer 1 until receiving an Ack message. Receiving a timeout signal of Timer 2, the protocol machine stops re-transmitting the message. In Figure 2, Timeout[1] and Timeout[2] denote timeout signals of Timer 1 and Timer 2, respectively.

### 2.3. FSM with timers

For a protocol machine \( M = (Q, X, n, Y, H, s_0) \) and a timer specification \( T \), if \( n \) equals to the dimension of \( T \), \( M \cdot T \) denotes the system consisting of \( M \) and \( T \). We use a pair \( \xi = (s, \vec{\tau}) \) of a state and a timer
value vector to describe an entire state of $M \cdot T$ where $s \in Q$ and $\bar{\tau} \in T$. Such a pair is called a composite state of $M \cdot T$. The set of all composite states of $M \cdot T$ is denoted by $Q_T$. The initial composite state of $M \cdot T$ is $\xi_0 = \langle s_0, \bot \rangle$, where $s_0$ is the initial state of $M$, and $x^n (x \in N \cup \bot)$ denotes a vector that all elements of the vector are $x$.

Given a composite state $\langle s, \bar{\tau} \rangle$, if there exists an integer $i$ (1 ≤ $i$ ≤ $n$) such that $\tau[i] = 0$ and $\tau[j] > 0$ for 1 ≤ $j$ < $i$, then $\langle s, \bar{\tau} \rangle$ is said to be timer $i$ expiring state. $\Gamma[i]$ denotes the set of timer $i$ expiring states. $\langle s, * \rangle$ denotes the set of composite states \{ $\langle s, \bar{\tau} \rangle \mid s \in Q, \bar{\tau} \in T$ \}. The timer value vector part of a composite state $\xi$ is denoted by $\tau(\xi)$. For the set of composite states $\Xi$, $\tau(\Xi)$ denotes the set of timer value vectors $\{ \tau(\xi) \mid \xi \in \Xi \}$.

We use $H_T$ to denote the set of composite state transitions of $M \cdot T$. $H_T$ is the set of composite state transitions $\eta = (\xi, \xi', x, y)$. There exist three kinds of transitions.

1. external input transitions:

   For an external input transition $(s, s', x, y, p) \in H$ there exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s', \bar{\tau}' \rangle, x, y)$ on $M \cdot T$ where $\tau[i] > 0 (1 \leq i \leq n)$ and

   $$\tau'[i] = \begin{cases} 
T[i] & (p[i] = S) \\
\tau[i] & (p[i] = N) \\
\bot & (p[i] = D) 
\end{cases}$$

2. timeout transitions:

   For a timeout transition $(s, s', \text{Timeout}[k], y, \tilde{p}) \in H$, there exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s', \bar{\tau}' \rangle, -, y)$ on $M \cdot T$ where $\langle s, \bar{\tau} \rangle \in \Gamma[k]$ and

   $$\tau'[i] = \begin{cases} 
T_i & (p[i] = S) \\
\tau[i] & (i \neq k \land p[i] = N) \\
\bot & (p[i] = D \lor (i = k \land p[i] = N)) 
\end{cases}$$

3. time elapse transitions:

   There exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s, \bar{\tau} - 1^n \rangle, -, -)$ on $M \cdot T$ where $\tau[i] > 0 (1 \leq i \leq n)$.

$H_t$ denotes the set of composite state transitions on $M \cdot T$ corresponding to $t \in H$. Also $H_e$ denotes the set of the time elapse transitions.

For a transition $\eta = (\xi, \xi', x, y), IO(\eta)$ denotes the input/output part of $\eta$, i.e. $IO(\eta) = (x/y)$. $\rho(\eta)$ and $\delta(\eta)$ denotes the source composite state and the destination composite state of $\eta$, respectively.

A composite state transition string is simply called a string. Given a string $r = \eta_1 \eta_2 \cdots \eta_m \in H_T^*$, $r$ is said to be executable from a composite
state $\xi_1 \in Q_T$, if there exist composite states $\xi_2, \ldots, \xi_{m+1} \in Q_T$ such that $\rho(\eta_k) = \xi_k$ and $\delta(\eta_k) = \xi_{k+1}$ for $k(1 \leq k \leq m)$.

Example 2 A string $\eta_1\eta_2\eta_3\eta_4\eta_5\eta_6$ is executable from the initial composite state of the protocol machine shown in Figure 2 where $\eta_1 \in H_{t1}$, $\eta_2, \eta_3, \eta_4 \in H_e$, $\eta_5 \in H_{t2}$ and $\eta_6 \in H_{t3}$.

We extend $\text{IO}$ and $\delta$ to accept a string or a set of strings. For a string $r$, $\delta(r)$ denotes the destination composite state of the last transition of $r$. $TS(\xi)$ denotes the set of strings which are executable from $\xi$, and $TS_n(\xi)$ denotes the set of strings in $TS(\xi)$ whose length are $n$. $R(\xi)$ denotes the set of composite states $\delta(TS(\xi))$. We may explicitly express the target machine $M$ in our notations, like $TS_M(\xi)$ or $R_M(\xi)$.

2.4. Related work

Timed automata[6, 7] are known as a model for describing and analyzing protocols which deal with time dependent behaviors. A timed automaton has several timers. The timer values increase by time elapse, and we can give timing constraints for transitions by simultaneous inequality on timer values. In state transitions, the timer value can be reset to zero. Although we do not give the proof, a timed automaton can simulate our FSM with timers. It means that the procedures for analyzing timed automata are also effective for analyzing FSM with timers.

3. TEST SEQUENCES DETECTING SINGLE FAULT

![Figure 3. Test architecture](image)

We consider a test architecture for FSM with timers (see Figure 3). In this architecture, the tester can measure time through timer system calls although the tester cannot observe and control the interactions between the protocol machine $M$ and timers. We assume that the tester receives
the timeout signals from timers after receiving all outputs from $M$. We use this assumption to construct test sequences in Section 3.4.

### 3.1. Fault model

We introduce a fault model for transitions of the protocol machines as follows.

- **timer command faults**:
  
  An element of timer command vector of a state transition differs from that of the specification.

- **destination state faults**:
  
  A destination state of a state transition differs from that of the specification.

- **external output faults**:
  
  An external output of a state transition differs from that of the specification.

In this paper, we focus to detect single faults. For a given specification $M = (Q, X, n, Y, H, s_0)$ and $T$, each faulty implementation can be denoted by $M[t'/t] = (Q, X, n, Y, H', s_0)$ and $T$ such that $H' = H \setminus \{t\} \cup \{t'\}$ where $t'$ differs from $t$ only in one element of the timer command vector, the destination state or the external output.

### 3.2. IO equivalent implementations

In a complex system, although an implementation $I$ does not conform to the specification $S$, a composed machine $I \cdot C$ ($C$ is a context) may conform to $S \cdot C$ [2, 3]. It is also true in our model of FSM with timers. It means that some implementations which do not conform to the specification cannot be externally distinguished from the specification in context. We introduce the following IO equivalent relation.

**Definition 1** Given two states $\xi_1$ of $A_T$ and $\xi_2$ of $B_T$, $\xi_1$ and $\xi_2$ are said to be IO equivalent, written $\xi_1 \equiv \xi_2$, if $IO(TS(\xi_1)) = IO(TS(\xi_2))$.

**Definition 2** $A_T$ and $B_T$ are said to be IO equivalent, written $A_T \equiv B_T$, if the initial composite states of $A_T$ and $B_T$ are IO equivalent.

Given a protocol machine $M$ and an implementation $M[t'/t]$ which contains a single timer command fault or destination state fault, $M[t/t]$ $T$ may be IO equivalent to $M \cdot T$. On the other hand, if an implementation $M[t'/t]$ which contains a single external output fault, $M[t'/t] \cdot T$ is not IO equivalent to $M \cdot T$ except the case that the faulty transition is not executable, since our tester can observe external outputs.
Example 3 For the protocol machine $M$ shown in Figure 4, let us consider a faulty implementation $M[t'_1/t_1]$ in which the transition $t_1$ is implemented as $t'_1$.

The initial state of $M$ is $\langle s_1, (\perp, \perp) \rangle$. In addition, after moving to $s_1$ by transition $t_3$, $(\tau_1, \tau_2) = (\perp, \perp)$. Thus, $(\tau_1, \tau_2) = (\perp, \perp)$ holds whenever $M$ is in $s_1$. It also holds for $M[t'_1/t_1]$.

Let us consider strings which are executable from $\langle s_1, (\perp, \perp) \rangle$ in $M$ and $M[t'_1/t_1]$. After executing $t_1$ or $t'_1$, $M$ is in $\langle s_2, (3, \perp) \rangle$ and $M[t'_1/t_1]$ is in $\langle s_2, (3, 8) \rangle$. For the both composite states, timer 1 will expire after three time elapse transitions. By the timer expiration, transition $t_2$ is executed, $M \cdot T$ is in $\langle s_3, (\perp, 8) \rangle$ and $M[t'_1/t_1] \cdot T$ is also in $\langle s_3, (\perp, 8) \rangle$. Furthermore, for the strings corresponding the above behaviors, $I O(\eta_1 \eta_2 \eta_3 \eta_4 \eta_5) = I O(\eta'_1 \eta'_2 \eta'_3 \eta'_4 \eta'_5)$ where $\eta_1 \in H_{t_1}$, $\eta'_1 \in H_{t'_1}$, $\eta_2, \eta_3, \eta'_2, \eta'_3, \eta'_4 \in H_e$ and $\eta_5, \eta'_5 \in H_{t_2}$. Thus $M \cdot T \equiv M[t'_1/t_1] \cdot T$.

3.3. Fault detecting sequence

For every implementation $M[t'/t]$, the equivalence between $M \cdot T$ and $M[t'/t] \cdot T$ can be decided by composing two finite automata which accept $I O(TS_{M[t'/t]}(\xi_0))$ and $I O(TS_{M[t'/t]}(\xi'_0))$ and deciding their equivalence. We can compose the above two automata in a similar way to the composition of the region automata from timed automata\cite{6}.

Since protocol machines are assumed to be deterministic and completely specified, if $M \cdot T$ is not IO equivalent to $M[t'/t] \cdot T$, there exist at least one IO sequence $io$ which is executable from the initial composite state of $M \cdot T$ and is not executable from that of $M[t'/t] \cdot T$. We can adopt such a sequence $io$ as a fault detecting sequence.

3.4. Translating IO sequences to test sequences

Let $io$ be an IO sequence obtained in the previous section. Since $io$ is merely an IO sequence of $M \cdot T$, we have to translate $io$ to a test
sequence observable from our tester. Since one or more consecutive timeout transitions occur after at least one time elapse transition, a string of \( M \cdot T \) should be an element of the set denoted by a regular expression \( \{(E^*F)|(E^+G^+)\}^*E^* \), where \( E \) denotes the set of time elapse transitions, \( F \) and \( G \) denote the set of external input transitions and that of timeout transitions, respectively. The translation is as follows.

- **IO sequences corresponding to \( E^*F \):**

  \[ (-/-)^n(x/y) \implies \begin{cases} (x/y) & n = 0 \\ (\text{Set}(n)/-)(\text{WE}/\text{TO})(x/y) & n > 0 \end{cases} \]

  
  Set\((n)\) means that the tester sets a timer to \( n \). WE means that the tester Waits Enough until an output comes. TO means a timeout signal corresponding to Set\((n)\). If the tester receives a timeout signal after setting a timer to \( n \), we can guarantee that \( n \) time units have passed.

- **IO sequences corresponding to \( E^+G^+ \):**

  \[ (-/-)^n(-/-)(-/y_1)\cdots(-/y_m) \implies \begin{cases} (\text{Set}(1)/-)(\text{WE}/y_1)\cdots(\text{WE}/y_m)(\text{WE}/\text{TO}) & n = 0 \\ (\text{Set}(n)/-)(\text{WE}/\text{TO}) & n > 0 \\ (\text{Set}(1)/-)(\text{WE}/y_1)\cdots(\text{WE}/y_m)(\text{WE}/\text{TO}) & n > 0 \end{cases} \]

  We treat \( E^+ \) as \( E^*E \). From the assumption, the tester can recognize the timeout signal for Set\((1)\) after observing all outputs \( y_1 \cdots y_m \) of timeout transitions.

- **IO sequences corresponding to \( E^* \):**

  \[ (-/-)^n \implies (\text{Set}(n)/-)(\text{WE}/\text{TO}) \quad n > 0 \]

**Example 4** For the protocol machine shown in Figure 2, suppose that the transition \( t_1 \) is implemented as \( t'_1 \).

\[
\begin{align*}
  t_1 &= (\text{Init}, \text{Wait}, \text{Send}, \text{Msg}, (S, S)) \\
  t'_1 &= (\text{Init}, \text{Wait}, \text{Send}, \text{Msg}, (N, S))
\end{align*}
\]

The following IO sequence is a test sequence detecting the fault.

\[(\text{Send}/\text{Msg})(\text{Set}(2)/-)(\text{WE}/\text{TO})(\text{Set}(1)/-)(\text{WE}/\text{Msg})(\text{WE}/\text{TO})\]

The specification accepts the above IO sequence, while the faulty implementation cannot accept it.
4. EFFICIENT TEST SEQUENCE GENERATION

In general, if we enumerate all faulty implementations, check their IO equivalence and generate the corresponding test sequences, the set of the test sequences can distinguish every faulty implementation with a single fault. However, this method may be an obstacle to generating test sequences rapidly. To reduce the time necessary for test sequence generation, we try to deal with multiple faults together and/or generate test sequences by analyzing a smaller FSM than $M \cdot T$.

Since external output faults can be immediately detected by executing the faulty transitions and observing the external outputs, we will focus on detecting timer command faults and destination state faults. We begin with enumerating the reachable states of $M \cdot T$ and generate strings which lead $M \cdot T$ from the initial composite state to each reachable composite state.

4.1. Test sequences for timer command faults

Assume that the specification is given as $M = (Q, X, n, Y, H, s_0)$. An faulty implementation $M[i/t']$ contains a single timer command fault for timer $i$ on a transition $t = (u, v, x, y, \bar{p}) \in H$.

We will discuss sufficient conditions for $M \cdot T \equiv M[i/t'] \cdot T$ and $M \cdot T \neq M[i/t'] \cdot T$ which can be checked by analyzing smaller machines. To consider such conditions, we introduce the following relation between composite states.

**Definition 3** Given two composite states $\xi$ and $\xi'$, we write $\xi <_{i} \xi'$ if $\xi$ and $\xi'$ differ only in the value of timer $i$.

For $\xi \in Q_T$ and appropriate transitions $\eta \in H_t$ and $\eta' \in H_{t'}$, $\delta(\eta) = \delta(\eta') = \xi$, $\delta(\eta) <_{i} \delta(\eta')$ or $\delta(\eta) = \delta(\eta')$ holds. Let us consider two composite states $\xi$ and $\xi'$ such that $\xi <_{i} \xi'$, a transition $z$ other than the timeout transition of timer $i$ and two transitions $\eta, \eta' \in H_z$ which are executable at $\xi$ and $\xi'$, respectively. If the timer command for timer $i$ of $z$ is either S or D, $\delta(\eta) = \delta(\eta')$ holds. On the other hand, if the timer command for timer $i$ of $z$ is N, $\delta(\eta) <_{i} \delta(\eta')$ still holds. It means that only strings consisting of the transitions corresponding to transitions whose timer command for timer $i$ is N can distinguish such $\xi$ and $\xi'$.

By the above consideration, we introduce a sub-protocol machine $M_N = (Q, X, n, Y, H_N, s_0)$ where $H_N = \{ z = (s, s', a, b, \bar{p}) \mid z \in H \land p[i] = N \land a \neq \text{Timeout}[i] \}$.

We introduce the following equivalence between strings.

**Definition 4** Assume that the sets of state transitions of $M$ and $M[i/t']$ are $H = \{ t, t_1, \ldots, t_k \}$ and $H' = \{ t', t_1, \ldots, t_k \}$, respectively. Two strings $q (= q_1 \cdot \cdot \cdot q_m)$ and $r (= r_1 \cdot \cdot \cdot r_n)$ of $(T_{M,T}(\xi_0) \cup T_{M[i/t']}, T(\xi_0))$ are said
to be \(\{t, t\}'\)-equivalent, written \(q^t \equiv_r T\), if (1) \(m = n\), (2) \(q_i \in (H_{t_j} \cup H_{t_j}') \Leftrightarrow r_i \in (H_{t_j} \cup H_{t_j}')\) for \(i(1 \leq i \leq m)\), (3) \(q_i \in (H_{t_j} \cup H_{t_j}') \Leftrightarrow r_i \in (H_{t_j} \cup H_{t_j}')\) for \(i(1 \leq i \leq m)\).

We extend the relation \(\equiv^{at, tm}\) to the relation between the sets of strings.

**Definition 5** Two sets of strings \(A, B\) are said to be \(\{t, t\}'\)-equivalent, written \(A \equiv^{at, tm} B\), if \(\forall a \in A \exists b \in B (a^t \equiv b) \land \forall b \in B \exists a \in A (b^t \equiv a)\).

Since composite state transitions are translated to IO sequences uniquely and composite state transitions corresponding to \(t\) or \(t'\) are translated to the same IO sequence, the following lemma holds.

**Lemma 1** For \(M \cdot T\) and \(M[t'/t] \cdot T\), if \(TS_{M \cdot T}(\xi_0) \equiv^{at, tm} TS_{M[t'/t] \cdot T}(\xi_0)\), then \(M \cdot T \equiv M[t'/t] \cdot T\).

In the following, we will consider a condition to decide \(TS_{M \cdot T}(\xi_0) \equiv^{at, tm} TS_{M[t'/t] \cdot T}(\xi_0)\).

For the strings whose length are one, the following lemma holds.

**Lemma 2** Given two composite states \(\xi\) of \(M \cdot T\) and \(\xi'\) of \(M[t'/t] \cdot T\), \(TS_{M \cdot T, 1}(\xi) \equiv^{at, tm} TS_{M[t'/t] \cdot T, 1}(\xi')\) if \(\xi = \xi' \lor \xi <_i \xi' \land (\xi, \xi' \notin \Gamma[i])\).

**Proof** Assume that \(\xi <_i \xi' \land (\xi, \xi' \notin \Gamma[i])\). Since the executability of external input transitions does not depend on the values of timers, if an external input transition is executable at \(\xi\), then the corresponding transition is also executable at \(\xi'\). Next, since \(\xi <_i \xi'\) and timeout transition caused by timer \(i\) is not executable at both \(\xi\) and \(\xi'\), the timeout transition which executable at \(\xi\) is \(\{t, t\}'\)-equivalent to that of \(\xi'\). Both the values of timer \(i\) at \(\xi\) and \(\xi'\) are greater than zero. Hence if the time elapse transition is executable at \(\xi\), then the transition is also executable at \(\xi'\). The case of \(\xi = \xi'\) can be shown in a similar way.

Table 1 describes a necessary and sufficient condition for \(TS_{M \cdot T}(\xi_0) \equiv^{at, tm} TS_{M[t'/t] \cdot T}(\xi_0)\). \(\Phi(u)\) denotes the set of timer value vectors at state \(u\) i.e. \(\tau(\mathcal{R}(\xi_0) \cap (u, *))\). For example, the first row denotes that \(M[t'/t] \cdot T\) cannot reach \(\Gamma'[i]\) after executing a transition corresponding to \(t'\) at \(\langle u, \tau \rangle\) (\(\tau \in \Phi_{M[t'/t] \cdot T}(u) \land \tau[i] \neq \bot\)).

**Lemma 3** The followings hold if and only if “condition” holds for every element \(\tau\) in “timer value vector” of the row according to the case of \(p[i]\) and \(p'[i]\) in Table 1.

1. \(TS_{M \cdot T}(\xi_0) \equiv^{at, tm} TS_{M[t'/t] \cdot T}(\xi_0)\).

2. For each \(\tau \in TS_{M \cdot T}(\xi_0)\) and corresponding \(\tau' \in TS_{M[t'/t] \cdot T}(\xi_0)\), \(\delta(\tau) = \delta(\tau')\) or \(\delta(\tau) <_{i} \delta(\tau') \land \delta(\tau), \delta(\tau') \notin \Gamma_i\).
Table 1. Necessary and sufficient condition for $TS_M \cdot T(\xi_0) \equiv' TS_{M[t'/t]} \cdot T(\xi_0)$

<table>
<thead>
<tr>
<th>$p[i]$</th>
<th>$p'[i]$</th>
<th>necessary and sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>N</td>
<td>$\tau [i] \neq \bot$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$</td>
</tr>
<tr>
<td>N</td>
<td>S</td>
<td>$\tau [i] = \bot$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$</td>
</tr>
<tr>
<td>D</td>
<td>N</td>
<td>$\tau [i] \neq \bot$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$</td>
</tr>
<tr>
<td>D</td>
<td>S</td>
<td>$\tau [i] = \bot$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$ $\forall \eta \in H \land \rho(\eta) = (\xi_0)$</td>
</tr>
</tbody>
</table>

The if part of Lemma 3 can be shown in induction using Lemma 2. By Lemma 1 and Lemma 3, if $M \cdot T$ and $M[t'/t] \cdot T$ satisfy the conditions shown in Table 1, $M \cdot T \equiv M[t'/t] \cdot T$ holds. The conditions in Table 1 can be checked by exploring a smaller FSM. Unfortunately, we have to compose the automaton for $M[t'/t] \cdot T$ in the case of $p[i] = S$ and $p'[i] = N$ even with Lemma 3.

In the case that the “condition” does not hold, a string $r$ which leads $M \cdot T$ or $M[t'/t] \cdot T$ from $(u, \tau)$ to a timer $i$ expiring state is obtained. By adding the transfer string from the initial state to $(u, \tau)$ in front of $r$, a string $r_0$ can be obtained. $r_0$ leads $M \cdot T$ or $M[t'/t] \cdot T$ from $\xi_0$ to a timer $i$ expiring state.

Next, we will consider a sufficient condition for $M \cdot T \neq M[t'/t] \cdot T$.

**Lemma 4** For $M \cdot T$ and $M[t'/t] \cdot T$, if there exist two $\{t, t'\}$-equivalent executable strings $r$ of $M \cdot T$ and $r'$ of $M[t'/t] \cdot T$, and if two composite states $\delta(r) = (s, \tau)$ and $\delta(r') = (s, \tau')$ satisfy either one of the following conditions, then $M \cdot T \neq M[t'/t] \cdot T$ holds.

- $(s, \tau) \in \Gamma[i] \land (s, \tau') \not\in \Gamma[i] \land \tau[j] > 0 (i < j \leq n)$.
- $(s, \tau) \not\in \Gamma[i] \land (s, \tau') \in \Gamma[i] \land \tau[j] > 0 (i < j \leq n)$.

**Proof** Suppose that one of the above conditions holds. If the value of timer $i$ is zero at $\delta(r)$ or $\delta(r')$, the values of other timers are not zero. Hence, an output caused by timeout of timer $i$ is observed.

Based on the above lemmas, we can summarize the test sequence generating procedure as follows.
step I According to the types of faults, check the condition in Table 1. If the condition holds, we do not generate a test sequence. Otherwise, we obtain the above mentioned string \( r_0 \).

If \( r_0 \) and corresponding \( r'_0 \) satisfy the condition of Lemma 4, we can adopt \( r_0 \) as a fault detecting string. Otherwise, go to step II.

step II By the way described in Section 3.3, we can obtain a fault detecting string.

4.2. Test sequences for destination state faults

Since the specification of protocol machines is assumed to be reduced, we can use existing test sequence generating methods, e.g. Wp-method [8], UIOv-method[9], to obtain strings which distinguish each state. We choose Wp-method for test sequence generation. However, if a string contains timeout transitions, the string may not be executable. Hence, we must decide whether the string is executable.

Given specification of a protocol machine \( M = (Q, X, n, Y, H, s_0) \), we consider to generate a test sequence for detecting destination faults on \( t = (u, v, x, y, \bar{p}) \in H \). We generate test sequences step by step. In the first step, we generate strings considering only external input transitions.

step 1 Find a state transition sequence which distinguishes \( v \) (the destination state of \( t \)) from \( w_1 \in (Q \setminus \{v\}) \) and contains only external input transitions in \( M \). If such a state transition sequence \( r \) exists, we can adopt a string \( Tr(u) \cdot \eta_1 \cdot R \) as a fault detecting string where \( \eta_1 \) is a composite state transition corresponding to a target transition \( t \), \( Tr(u) \) is a transfer string which leads \( M \cdot T \) from the initial composite state to a composite state \( \langle u, \bar{r} \rangle \) in which \( \eta_1 \) is executable, and \( R \) is a string corresponding to \( r \).

step 2 Let \( w_2 \in (Q \setminus \{v\}) \) be a state which cannot be distinguished from \( v \) by any string consisting of external input transitions. Find a state transition sequence \( t_1t_2 \cdots t_n \in H^* \) which distinguishes \( v \) from \( w_2 \) and decide whether there exists a string in \( H_e^* H_{t_1}^* H_e^* H_{t_2}^* \cdots H_e^* H_{t_n} \) executable in \( M \cdot T \). Recall that \( H_e \) is the set of time elapse transitions. If such a string exist, we can generate the fault detecting string by adding some appropriate transfer string.

step 3 If any string in \( H_e^* H_{t_1}^* H_e^* H_{t_2}^* \cdots H_e^* H_{t_n} \) is not executable, there must be unexecutable timeout transitions \( t_k \) in the string. So as shown in Figure 5, if timer \( i \) is active in some composite state \( \langle s_k, \bar{\tau} \rangle \) and there exists a cycle \( c_1 \cdots c_m \) consisting of transitions whose commands for timer \( i \) is N, then we try to add the cycles in front of the unexecutable transition \( t_k = (s_k, s_{k+1}, \text{Timeout}_i, y, \bar{p}) \).
Figure 5. String extension

We also check whether there exists a string in \( H_e^* H_{t_1} \cdots H_e^* H_{t_{k-1}} \{ H_e^* H_{c_1} \cdots H_e^* H_{c_m} \}^+ H_e^* H_{t_k} \cdots H_e^* H_{t_m} \cdot \) executable in \( M \cdot T \).

**step 4** For a state \( w_3 \in (Q \setminus \{v\}) \) at which we cannot find a string in the above steps, we can obtain a fault detecting string in the way described in Section 3.3.

In step 1, only a specification \( M \) is analyzed. In step 2 and step 3, a specification with timers \( M \cdot T \) is analyzed. In step 4, the system consisting of a faulty implementation and timers \( M[t'/t] \cdot T \) is analyzed.

5. **AN EXPERIMENT**

We developed a test sequence generating system based on the proposed method, and applied the system to DHCP (Dynamic Host Configuration Protocol) [10].

5.1. **Using simultaneous inequality**

In composing a finite automaton of \( M \cdot T \), the state space explosion problem arises for protocols containing timers with large \( T[i] \) values, e.g. two or more \( T[i] = 1000 \) timers.

To avoid this problem, we introduce simultaneous inequality to express the set of timer value vectors. The inequalities are classified into the following two types; (i) an inequality \( \tau[i] \leq C \) or \( \tau[i] \geq C \) specifies the upper limit or the lower limit \( C \) (a constant value) of timer \( i \), (ii) an inequality \( \tau[i] - \tau[j] \leq C \) specifies the upper limit \( C \) of the difference of two timer values \( \tau[i] \) and \( \tau[j] \). Simultaneous inequality consisting of the above types’ inequalities can be solved in \( O(lm) \) (\( l \): the number of inequalities, \( m \): the number of variables) [11]. For using such simultaneous inequality, we have made a small modification on our test sequence generating procedure.

A similar method is introduced in verification of timed automata [7].

5.2. **A sample protocol**

DHCP provides a framework for passing configuration information to hosts on TCP/IP networks. DHCP is built on a client-server model,
where designated DHCP servers allocate network addresses and deliver configuration parameters to clients dynamically. A client requests the use of an address for some period of time (lease time). Both the client and server use timers to watch the lease time. In DHCP, the time unit is a second. The shortest lease time is an hour. A client re-transmits a message based on timeout. The first interval of re-transmitting is four seconds. The interval becomes twice every re-transmission.

If the lease time can be specified by a client, we cannot describe DHCP in our model. Hence, we limit the lease time to an hour, i.e. 3600 time units. In our model, the number of states is 11 and the number of state transitions is 74. The number of timers is nine where five timers are used to watch the interval of re-transmitting messages. The rest of the timers are used to watch the lease time.

### 5.3. The result

As a result, we can decide the IO equivalence between the specification and faulty implementation, and generate test sequences for non IO equivalent implementations in a reasonable processing time. The experiment is done on a PC (CPU : Pentium III 600MHz, Memory : 128MB). The computing time and memory space are about seven minutes and 2MB, respectively.

In Table 2, we show the total number of the implementations containing a single fault (test item), the number of IO equivalent implementations, and that of generated test sequences (non IO equivalent implementations). These tables also tell the numbers of faulty implementations shown to be IO equivalent or generated test sequences in every step, which correspond to each step described in Section 4.

| step | timer command | test item | 1332 | equivalent | 457
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>step I</td>
<td>test sequence</td>
<td>875</td>
<td></td>
<td></td>
</tr>
<tr>
<td>step II</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| step | destination state | test item | 740 | equivalent | 0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>step 1</td>
<td>test sequence</td>
<td>740</td>
<td></td>
</tr>
<tr>
<td>step 2, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>step 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For every timer command faults, the equivalence was determined and the test sequence was generated for non IO equivalent fault in step I. About 34% of faults are equivalent to $M\cdot T$. Most of these faults are the one that the timer command $N$ for non-active timers at source states of target transitions are implemented as $D$.

For the destination state faults, about 95% of the test sequences are generated in step 1, and no test sequence is generated in step 4.
There is no test sequence which is generated by enumerating all reachable states. Test sequences for DHCP are generated efficiently.

6. CONCLUSIONS

In this paper, we have proposed a method to generate conformance test sequences for FSM with timers. The test sequences generated by this method can detect any single fault of timer commands or destination states. We have developed a test sequence generating system and applied it to DHCP. As a result, the test sequences was generated efficiently.

Acknowledgments

This work is partially supported by International Communications Foundation (ICF), Japan.

REFERENCES

Part Seven

Formal Methods in Software Development II
This page intentionally left blank.
A TOOL FOR GENERATING SPECIFICATIONS FROM A FAMILY OF FORMAL REQUIREMENTS

Jan Bredereke

Universität Bremen, FB 3 · P. O. box 330 440 · D-28334 Bremen · Germany
jan.bredereke@web.de · www.tzi.de/~brederek

Abstract. We are concerned with the maintenance of formal requirements documents in the application area of telephone switching. We propose a specification methodology that avoids some of the so-called feature interaction problems from the beginning, and that converts some more into type errors. We maintain all the variants and versions of such a system together as one family of formal specifications. For this, we define a formal feature combination mechanism. We present a tool which checks for feature interaction problems, which extracts a desired family member from the family document, and which generates documentation on the structure of the family. We also report on a quite large case study.

Keywords: tools; formal requirements; maintenance; feature interaction problems; telephone switching; Object-Z.

1 INTRODUCTION

Digital telephone switching systems already comprise hundreds of features, and the market forces the providers to incorporate an ever increasing number of new services and features into the switches. In recent years, feature interaction (FI) problems have become a serious obstacle to adding more features to these systems [1–3]. Adding one more feature, even when it behaves as desired for itself, may cause undesired behaviour of other features, even when these behave as desired for themselves. The number of possible feature combinations has become so large that it is not possible anymore to check all combinations manually.

One standard example of such a feature interaction occurs between the Terminating Call Screening (TCS) and the Call Forwarding (CF) feature (Fig. 1). Terminating Call Screening allows a subscriber C to specify a list of callers by whom he never wants to be called. Call Forwarding allows a subscriber B to specify another destination to which all calls to B should be forwarded. Now suppose that C has subscribed to TCS, and has put A on his screening list. Furthermore suppose that B has subscribed to CF and forwards all calls to C. Finally, A happens to attempt to call B. Will the phone of C ring? Should it ring? This depends very much on how we define the notion of “caller” for the two features. When CF consists of two calls “glued” together, then B is the caller for C, the screening condition is not satisfied, and the phone will ring. When we take into account that C probably does not want to be disturbed by A, the phone should not ring. We will come back to this example later.

The point we want to make at the above example is that feature interactions already arise in the requirements documents. When these documents are a complete description of the behaviours (and of other interesting properties), then even all feature interaction
problems are at least inherently present in the requirements documents. Therefore, they should be tackled already there.

The notion of complete requirements leads us to formal, i.e., precise, requirements. Considerable work about formal requirements specifications exists. But an aspect of them that deserves more attention is the maintenance of formal requirements documents. How can we handle many changes to them over time, and how can we handle a large number of variants for these documents when different customers of a successful product have different needs?

In a recent paper[4], we introduced the idea of families of formal requirements. Our approach proposes to keep all requirements variants together, and to encapsulate variant choices. It enables information hiding for requirements documents, similar to how it is known for programs. Furthermore, it encourages to plan ahead for future members of the family. Another core idea is to let the original specifier document explicitly as much of the information as possible that another specifier of an incremental feature will need. The specifier of an incremental feature can use this explicit information either for avoiding feature interaction problems, or for at least detecting feature interaction problems, such that they can be resolved.

A family of requirements needs a means for incremental specification. One standard means is refinement. In refinement approaches, a base system is specified first, which is then extended step by step by specifying further parts. The formalisms are constructed such that the interesting properties of previous steps are preserved by subsequent steps. For example, some approaches allow to add further possible behaviour to a behaviour description, while all existing behaviour remains possible. Other approaches allow to impose incremental, explicit constraints on an (implicit) behaviour description, which preserves all previous constraints due to the nature of the logical “and”. While these formal mechanisms are useful for many applications, they are not yet sufficient in our application area of telephone switching. In our experience with telephone switching requirements, most new features change the behaviour of the base system or of other features. “Change” here means that something really becomes different than it was before. For example, the Call Forwarding feature stops to connect to the dialled number and thus restricts the base system behaviour, and it starts connecting to the forwarded-to number, extending the base system behaviour. A telephony feature typically is non-monotonous.

Various feature operators have been proposed to allow non-monotonous changes. They allow to express the desired changes. If their definition and use reflect the current practice of modelling telephone switching features, feature interaction problems occur in such a formal setting in the same way as in an informal setting. A lot of research work has been done on a formal analysis of the changes [1–3, 5]. Our experience is that this indeed works to some extent, but that the exponential growth of complexity not
only beats the human specifiers, but also beats the automated tools, when it comes to real-sized systems with hundreds of features. In particular, feature operators allowing arbitrary syntactic modifications of individual source lines lead to unmanageable feature interaction problems, since the complete system must be analyzed for the consequences.

We therefore attempt to reduce the complexity by using modularization in the information hiding sense for the requirements documents. We support it by a feature construct which allows for non-monotonous changes but which does not allow for arbitrary changes. In our experience, planning ahead for a careful modularization and for future potential modifications reduces the number and complexity of the dependences between features.

In Section 2, we briefly introduce to the formal specification language we use, and we present a suitable, new feature construct for it. Section 3 describes our tool support for managing features. Section 4 demonstrates the application of the approach and the tool in a case study from telephone switching, and Section 5 discusses how we avoided or detected feature interaction problems. Section 6 compares our approach to related work, and Section 7 summarizes this paper.

2 A SPECIFICATION METHODOLOGY SUPPORTING FAMILIES OF FORMAL REQUIREMENTS

In this section, we give a very brief overview of the formal specification language we use, concentrating on the means to structure a specification document. Another prerequisite is a suitable specification style, which we describe. We then add in a first step the concept of families to this language, and in a second step the concept of non-monotonous, but well-structured, changes.

2.1 Base Language CSP-OZ

We have chosen the formal specification language CSP-OZ [6,7] as a base for adding our feature constructs. This language is suitable for telephone switching, since it supports communication events explicitly, and it is is suitable for specifying requirements, since it allows to specify the communication events that are observable externally, without prescribing any internal structure. CSP-OZ is a combination of the process algebra CSP (Communicating Sequential Processes) [8] and of Object-Z [9]. CSP-OZ allows to specify communication and control aspects in the CSP part, and data aspects separately in the Object-Z part. Such a separation has proven useful in the communications area. The Object-Z data part is in turn based on the language Z [10, 11], which is a formalization of set theory. Object-Z provides the notions of abstract data types and state invariants which are not present in CSP. Numerous algebraic laws have been derived for the refinement of behaviour in CSP and for the refinement of data in Z, providing a sound theoretical base to the inheritance operators we use.

The inheritance operator of CSP-OZ, in its basic form, allows to specify refinement. Optionally it can be supplemented by a renaming operator. If the latter is used, it effectively allows to change the behaviour of an inherited class completely. CSP-OZ inheritance thus offers either pure refinement or powerful, but uncontrolled, changes.
The Z language was defined for many years by Spivey’s reference manual [11]. Currently, the International Standardization Organization (ISO) finalizes a standard for Z [10]. Besides providing a more detailed, well-structured definition of the syntax and semantics of Z, the standard also provides a few extensions to Spivey’s Z. One of these extensions concerns the structuring of a specification document, and it supports our work considerably. We therefore already base our work on the forthcoming Z standard. Fortunately, the definition of CSP-OZ is also already based on the new Z standard.

A Z (and thus also a CSP-OZ) specification consists of paragraphs of formal text. A paragraph can be, for example, one type definition or one schema definition. The new standard allows to put a specification and sections on top of paragraphs. A specification consists of sections, which in turn consist of paragraphs. The meaning of a Z specification is the set of named theories established by its sections. Each section adds one named theory to the set established by its predecessors. A named theory associates a section name with a set of models.

As a consequence, each section has a self-contained formal meaning. Any initial sequence of sections can be taken as the requirements for a variant of the specified system. Since each further section can only add more constraints on the system (and new declarations), this allows for a constraint-oriented, incremental style of specification.

But there is no means for non-monotonous extensions. Z sections have a parents construct such that a section can be an extension of specifically named sections. The only way to get a modified version of a system is to copy, rename, and edit the changed sections, and also to copy and rename all sections that have these sections as a parent. If a section needs to be modified that is at the leaves of the hierarchy of parent sections, this can amount to a large part of the document being duplicated.

2.2 Specification Style

For requirements, we prefer a constraint-oriented specification style. Adding one (small) constraint after the other helps us to concentrate on one aspect of the system at the time, without accidentally being overly restrictive on other aspects.

When we perform non-monotonous extensions, it is important that the changes happen in a controlled way. Therefore, we choose that a feature may modify another feature at the level of Z sections. Leaving out an entire feature and introducing a new feature with a slight modification is too coarse-grained. If the feature is complex, this results in a considerable duplication of code. Allowing a feature to modify the specification text of another feature in an arbitrary way, e.g., by changing an integer value or by substituting an arithmetic operator, is too fine-grained. It is hard to understand the consequences of such changes, in particular if many of them are applied at the same time. This easily leads to feature interaction problems.

We distinguish between the essential behaviour and the changeable behaviour of a feature. Some parts of a feature are essential for its nature. Other parts are provided only in order to make the requirements specification complete. For example, some of the behaviour restrictions might only be made to make the behaviour predictable for the user. If the changeable behaviour is modified, this can never be an undesired feature interaction. When a specifier needs to modify a feature that was written some time ago by
another person, he will find it difficult to find out which part of the behaviour is essential. Therefore, we require the original specifier of a feature to document which behaviour is essential and which behaviour is changeable, by using an appropriate language construct.

Furthermore, we distinguish between the requirements of the system and the requirements of its environment. This is important for any software development contract. It is the developer’s duty to implement the former, and it is the customer’s duty to ensure the latter, in order to make the system actually work. Formally, we collect both parts into one CSP-OZ class each, using inheritance. We then compose them by parallel composition into a description of the aspects of the world which are relevant to the contract.

2.3 Extension of CSP-OZ: Families with Monotonous Increments Only

In order to support families, we change the top-level structure of the language. We remove specifications from the syntax, and we add families, feature chapters, and family member chapters instead.

A family consists of feature chapters and of family member chapters. Feature chapters are (named) chapters of (named) sections. Family member chapters are named chapters that contain nothing but a features list. Informally, each family member chapter defines one specification in plain CSP-OZ. We can extract one family member from a family document by collecting all its features, and by concatenating all the sections of these features, resulting in a specification in plain CSP-OZ.

A detailed definition of the syntax and semantics of the language extension can be found in the manual [12].

The inheritance operator of standard CSP-OZ can be supplemented by a renaming operator, effectively allowing to change the behaviour of an inherited class completely. We therefore ban the use of the renaming option in our extension of CSP-OZ.

Semantics. The set \( \text{Features} \) contains all feature chapters’ names. The set \( \text{Sections} \) contains all mappings from feature chapter names to sets of section names: \( \text{Sections} = \text{Features} \rightarrow\{\text{Name} \rightarrow \text{P Name} \} \). The meaning of a feature chapter can be determined by applying the function \( \text{Sections} \) to the feature chapter’s name. The meaning of a family is a mapping from family member names to meanings of plain CSP-OZ specifications.

\[ [\cdot]^F : \text{Family} 
\rightarrow (\text{Name} \rightarrow \text{P Theory}) \]

That is, each (pure) family member chapter is essentially a complete CSP-OZ specification, except that all of its actual sections have been moved into feature chapters, which can be shared with other family members.

Shorthand Notations. Optionally, a family member chapter may contain a feature chapter body. This is transformed syntactically into a separate feature chapter and a “pure” family member chapter. Such a feature chapter body can be convenient to specify the composition of all items from the different features in a suitable way.

CSP-OZ classes are larger units than mere (Z language) paragraphs. Therefore we found it sometimes convenient to use classes for structuring instead of sections. This
is true in particular when each section contains only one class. Therefore, we allow a class in the places where a section can occur, too. When determining the semantics, we transform such a class into a section heading followed by this class, and followed by whatever other formal paragraphs may follow. The name of this implicit section is the name of the class. (Please note that family members, features, and sections each have separate name spaces.) The parents sections names of the section are the names of the classes inherited by this class.

2.4 Extension of CSP-OZ: Families with Non-Monotonous Increments

Sections are our unit of increment for monotonous increments, and they are also our unit of modification for non-monotonic changes. We distinguish between the essential and the changeable behaviour of a feature, and we specify this information explicitly by using different kinds of section. The type rules on the new constructs defined below will allow us to exploit the information for formal checks.

But one more kind of section is not sufficient. Both the essential and the changeable behaviour constraints need to be composed. The composition operators need to be grouped into sections too, and these operators specify neither essential nor changeable behaviour. They require a third kind of section.

We therefore add two more kinds of section. We put the essential properties of a feature into the normal kind of section. The two new kinds of section have the same syntax as the normal kind, except for the new keywords default_section and collect_section. Default sections are different insofar that they can be referenced by a remove clause, and they serve to express changeable properties. A remove clause is a another new construct that may appear in feature chapters. When the family member comprises a feature that contains a remove clause, the named default sections are disregarded when determining the set of sections. A collect section is special in that it adapts its list of parent sections automatically to losses of default sections due to remove clauses. A collect section should be used only to collect and combine properties defined in other sections.

We define six type constraints on these different kinds of sections which ensure their intended use, and which flag errors that may constitute feature interaction problems. A basic rule is that only changeable properties can be removed: 1) A remove clause may name default sections only. 2) The parent of a normal, i.e., essential, section must not be a default section, except if it is from a different feature or family member chapter. The above exception is necessary to allow one feature to use a specific, non-essential version of another feature as a base. Note: a type checker may issue a warning if there is more than one remove clause for one default section in the features of a family member.

By further rules, parents clauses and remove clauses are restricted to sensible section names with respect to any specific family member: 3) For each family member \( m \), we construct the set \( \text{SectionNames}(m) \) of names of all its feature’s normal sections, default sections, and collect sections. For each family member \( m \), all remove clauses of its features must name sections from the set \( \text{SectionNames}(m) \) only. 4) We also construct a reduced set \( \text{RSectionNames}(m) \) from \( \text{SectionNames}(m) \) by removing those default sections that are named in a remove clause of this family member. For any normal or default section in \( \text{RSectionNames}(m) \), its set of parents sections must be a subset of \( \text{RSectionNames}(m) \).
Finally, the varying set of defined items in collect sections cannot be referenced in property-defining sections: 5) Neither normal nor default sections may have a collect section as a parent. 6) The parents lists of collect sections must contain sections only that are either from the same feature or that are collect sections.

We also introduce a special kind of inheritance operator, called “default_properties”, that is ignored in case the inherited class has been removed. We extend the syntax accordingly, and we add two more type constraints for the new inheritance operator: 7) The default properties operator may appear in a collect section only. 8) The class name in the operator must be a valid class name in the entire family document; but for any specific family member, the default properties operators in its sections may reference class names from outside of these sections.

A detailed definition of the syntax and semantics of this part of the language extension can be found, again, in the manual [12].

**Semantics.** The meaning of a remove clause is the set of its names of default sections. The function Remove maps feature chapter names to the sets of names of default sections which are named in a remove clause in this feature chapter: Remove : Features \(\Rightarrow\) \(\mathbb{P}\) Name.

The features list of a family member chapter determines the features that are used to construct its CSP-OZ specification. The features list is a subset of the set Features. Its meaning is the union of all the sets of section names of the features listed, minus the set of default sections that are removed by the features listed.

\[
[fe_1 \ldots fe_n]\_{FL} = (\text{Sections}(fe_1) \cup \ldots \cup \text{Sections}(fe_n)) \setminus (\text{Remove}(fe_1) \cup \ldots \cup \text{Remove}(fe_n))
\]

The sections of a family member chapter’s specification are all the sections \([fe_1 \ldots fe_n]\_{FL}\). The only modification is that we remove all section names in (Remove \( fe_1 \cup \ldots \cup \text{Remove} fe_n\)) from the sets of parents sections. The meaning of the specification is then constructed in the usual way from these sections.

**Shorthand Notations.** When classes are used for structuring instead of sections, a class is transformed into a collect section instead of a normal section exactly if it contains a “default_properties” operator. A class is transformed into a default section exactly if there exists a “default_properties” operator somewhere in the same feature that references its name.

### 3 THE TOOL

Our approach is supported by a tool, called genFamMem 2.0, which

- extracts specifications in plain CSP-OZ from a family document,
- detects feature interactions by
  - performing the additional type checks for families like the constraints on normal, default, and collect sections, on “remove”, and on the “default_properties”/“inherit” operators,
Fig. 2. Structure of the genFamMem tool.

- issuing some heuristic warnings, for example when two different remove clauses remove the same class, and
- helps avoiding feature interactions by generating documentation on the structure of the family.

The tool contains a lexer/parser for our extension of CSP-OZ which is written using the lex/yacc parser generator [13], and C code. The tool comprises about 8500 lines of commented source code. Figure 2 shows its module structure. The tool has a modular structure that reflects the well-structured definition of the Z semantics in the forthcoming standard. The standard defines the syntax and the semantics in nine phases, the first ones being mark-up, lexing, and parsing, and the last one being the semantic relation. The separate mark-up phase solves the problem that Z provides many more special symbols than normal keyboards can provide, by encoding them as sequences of characters. The standard defines two mark-ups, L\textbackslash T\textbackslash E\textbackslash X markup and email mark-up.

The mark-up module takes a family document for a first pass and translates L\textbackslash T\textbackslash E\textbackslash X control sequences into CSP-OZ characters like “\textbackslash F”, “\rightarrow” etc. It could be replaced by mark-up modules for other mark-ups. The next steps are the lexical and the syntactic analysis. The semantic calculation module takes the information from the syntactic analysis, and the name of a family member, and it calculates which sections should be included into the generated specification. It also performs the type checks described above, and flags any type errors in the family document structure, indicating feature interaction problems. When no type errors occurred, the tool extracts the actual specification document in a second pass, while adjusting the parents clauses, turning the default properties clauses into appropriately reduced inherit clauses, and generating the
The uses-hierarchy of the sections of feature UserSpace from the case study [16, 17].

The uses-hierarchy of family member Specification C from the case study, abstracted to features.

Besides generating and checking family members, the tool genFamMem also has another mode of operation, in which it generates documentation on the structure of the family. It extracts the parents relations of all sections and the inheritance relations of all classes, it extracts the association of sections to features, and then it constructs and outputs a uses-hierarchy graph. The uses-hierarchy graph describes which section needs which sections as prerequisites. This graph then is visualized graphically using the tool daVinci [15]. We have the choice to see the uses-hierarchy of all sections of the family (which is large!), of one family member only, or of one feature only. Furthermore, we have the choice to abstract sections to features, such that we see the uses-hierarchy of all features of the family, or of all features of one family member.

For example, Figure 3 presents the uses-hierarchy graph of the sections of feature UserSpace from our case study [16, 17]. Plain grey boxes denote essential sections, white boxes denote changeable sections, and double border boxes denote collect sections. Since the uses-relation is transitive, redundant arrows are not shown, such as the one from DevAssocNoRem to HumanEss. Figure 4 shows the uses-hierarchy of family member Specification C from our case study, abstracted to features.
Our extensions can be defined not only for CSP-OZ, but also for plain Z and for CSPZ. The latter is a dialect of CSP that uses Z for data definitions, but does not offer object-based constructs like classes and inheritance. Our tool already supports Z and CSPZ through switches which optionally disable the more advanced language constructs. We have not conducted experiments with families of Z or CSPZ specifications yet, but plan to do so.

The tool is available without charge through its Web home page [18].

4 CASE STUDY

We specified the requirements for a telephone switching system in a case study [16, 17], using our extension of CSP-OZ. The case study currently comprises about 40 pages of commented formal specification, with about 50 sections in nine features, including the base system. We demonstrate the new structuring means using excerpts from two simple features and from the base switching system.

The family of requirements from the case study can be used to generate requirements specifications with different combinations of features. Such a requirements specification can then be the starting point for a software development process delivering an implementation for this family member.

4.1 The Base Telephone Switching System

The base concepts of telephony are at the bottom of the class inheritance hierarchy. The class HumanEss introduces the set of humans, modelled as the variable humans. Classes are written in CSP-OZ as a box, with the class name in the heading. We write the state space of the class in a contained, unnamed box. The (essential) class UserEss introduces the set of user roles users of humans and also the association of humans to user roles userRoles:

```
section HumanEss [ Humans ]
  parents
  ...
  HumanEss
    ...
  humans : ↦ humanlds
```

```
section UserEss [ User Roles ]
  parents HumanEss
  ...
  UserEss
    inherit HumanEss
    ...
  users : ↦ userlds
  userRoles : humanlds → ↦ userlds
```

The class DeviceEss introduces the set of terminal devices devices and also the association of humans to devices possessedDevices. The section and class DeviceEss is very similar to UserEss, we do not reproduce it here.
In order to get a plain old telephone system, we specify next that the elements of these sets are all in a one-to-one relation, and that these sets are static. For example, the association between humans and devices should never change:

\[
\text{default_section DevAssocNoRem \{ Telephone Device Associations neither Added nor Removed \}}
\]

\[
\text{DevAssocNoRem}
\]
\[
\text{inherit HumanEss, DeviceEss,} \ldots
\]

\[
\forall h : \text{humans} \cdot \text{possessedDevices} (h) = \text{initialPossessedDevices} (h)
\]

The specification describes furthermore how a connection is set up and torn down. Due to space limitations, we leave out this rather large description \cite{16, 17} here.

### 4.2 A Black List Feature

A black list feature protects its subscriber from calls of users that she has put on a black list. We specify the black list feature in two steps. First, we specify a feature \text{ScreeningBase} that only serves as a "hook" which allows to block connections according to some criteria. The variable \text{connectionScreen} is defined in section \text{ConScreenEss}, it describes the pairs of users which shall never be connected. We specify this screening as a kind of missing resource for a connection. A default section \text{ConNoScreen} of this feature then states that the screening condition is never satisfied.

In a second step, we specify one specific kind of black list feature \text{BlackListOfUsers}, including means to add users to the list and to delete them again. Due to the size of this definition \cite{16, 17}, we omit it here. But we present how we couple the feature \text{blackListUsr} to the general screening above:

\[
\text{feature BlackListOfUsers}
\]

\[
\text{... remove ConNoScreen}
\]

\[
\text{default_section BlackListUsrOnly \{ User Black List is the only Screening Feature \}}
\]

\[
\text{BlackListUsrOnly}
\]
\[
\text{inherit \ldots, ConScreenEss}
\]

\[
\forall u, v : \text{users} \cdot ((\text{connectionScreen} (u, v) = \text{connectionScreenBlock}) \iff ((u, v) \in \text{blackListUsr}))
\]
4.3 A Call Forwarding Feature

The feature FollowHumanConnectionForwarding allows a human to move to another telephone device and register himself there, in order to have all of his calls forwarded there. We specify this feature mainly by lifting the restriction of a static association between humans and devices introduced above.

feature FollowHumanConnectionForwarding

... remove DevAssocNoRem

In order to facilitate the implementation of these requirements, the feature also demands that a registration may be performed only when no connection to the human exists. Again, we omit the details [16, 17] due to space limitations.

4.4 Composition of a Complete System

The above two features, and some more, can be combined into the specification of the requirements of a complete system:

familymember SpecificationC

features ..., BlackListOfUsers, FollowHumanConnectionForwarding

The collect sections of the listed features are composed in a single collect section of the family member. The structure of this collect section always follows a fixed pattern: one class inherits all requirements on the system, a second class inherits all requirements on the environment, and in a third, final class the first two classes are composed by a parallel composition operator:

main = SpecificationCEnv | SpecificationCSys

Since this final collect section has such a fixed structure, it can be omitted, the semantics definition of the language then derives it from the features list. The entire family member chapter then consists of the features list only. Accordingly, our genFamMem tool fills in this collect section when generating a family member in plain CSP-OZ.

Similarly, we do not need to specify the parents sections of a section under certain circumstances, they are then derived from the inheritance operators. We omit the formal details [12] of these shorthand notations here.
Performing the Generation. We use the tool genFamMem to extract the above family member from the family document. The command

```
genFamMem SpecificationC case-study. tex > SpecificationC. tex
```

extracts `SpecificationC` and performs the type checks, and the command

```
genFamMem --uhier-familymember SpecificationC --only-features \ case-study.tex > SpecificationC.daVinci
```

extracts the uses-hierarchy graph shown a few pages above in Figure 4.

5 AVOIDING AND AUTOMATICALLY DETECTING FEATURE INTERACTIONS

The goal of our approach is to either 1) avoid feature interaction problems in requirements documents from the beginning, or 2) to help detect them, such that they can be resolved, by

- allowing to maintain all variants of a requirements document together, by
- using a specification style that encapsulates variant choices, and by
- allowing/urging the original specifier to document as much of the information as possible that another specifier of an incremental feature will need.

The information provided formally by a specification written in CSP-OZ with our extension comprises

- the *dependencies* among the features, through the hierarchy of parents sections sets and of class inheritance, and
- what is *the core of a feature*, by the distinction between default sections, which are changeable, and normal sections, which are essential.

This information can help the maintenance specifier to avoid problems, and it can be used for automated type checks, as performed by our generator tool. The detected feature interactions then can be resolved by the specifier.

Therefore, thanks to the additional explicit information in the document, our approach converts some of the feature interaction problems into *type errors*.

5.1 Example of Feature Interaction Avoidance

In our case study, we explicitly introduce the notions of telephone devices, humans, and user roles, even though we then state a static association between these for the Plain Old Telephone Service. Nevertheless in the definition of the screening feature presented above, we consequently must specify that user roles are screened, not devices. Furthermore, we encapsulated this variant choice by specifying separately a screening base feature and a user screening feature. We also specified another variant, a device screening feature not presented in this paper, which also builds on the screening base feature.

Similarly, our case study comprises two kinds of call forwarding features. One registers a human with another device, it is presented above. The other feature transfers a user role to another human. An example use case would be a help desk employee who transfers
his help desk user role to a colleague while having a coffee break. His boss can still reach him in his role as an employee, even though he is not disturbed by customers.

We can now come back to the standard example of a feature interaction between Terminating Call Screening (TCS) and Call Forwarding (CF) from the introduction. Due to our separation of notions, for each combination of forwarding/screening features it is clear whether the phone should ring. A subscriber of BlackListOfUsers will be protected against a connection from a screened user from all devices, at the device where she is registered currently. A subscriber of BlackListOfDevices will not expect to be protected from humans that are able to use different devices. We just don’t have the ambiguous notion of “caller” anymore.

5.2 Example of Feature Interaction Detection and Resolution

Our case study does not have any feature interaction problems between Terminating Call Screening and Call Forwarding, and therefore we cannot demonstrate any feature interaction detection using this standard example. Nevertheless, our type rules defined above point us to some remaining problems.

For example, both screening features “remove” the same changeable section ConNoScreen of the screening base feature (compare Section 4.2, first paragraph). Each feature then states in one new, essential section when no connection must be possible. The type rules flag this double removal as a warning. A manual analysis shows immediately that the two replacing, simple sections are not contradictory. But a further look reveals that both features also have a changeable section each that states that its respective feature is the only screening feature, such that otherwise no screening happens. This of course is a contradiction in the general case.

After we have detected a feature interaction problem, we need to resolve it. As in most cases, the harder part was to detect the problem before a customer notices it, and the relatively easy part is to find a specific remedy. We specify this remedy as just another feature which we select whenever both of the above features are included. It contains a section that states that screening happens exactly when either of the two screening features demands it. Furthermore, it removes both conflicting sections. We are explicitly allowed to do this since both sections are marked as changeable.

5.3 Conventional Further Consistency Checks

After we have generated one specific requirements document, all the conventional methods and tools can be applied which check the consistency of formal requirements. We used the CSP-OZ type checker cspozTC [14] to find conventional type errors (“bugs”), but they are of no interest here.

6 RELATED WORK

Related work investigates “feature” constructs for different specification and programming languages, in particular in the context of the recently terminated FIREworks project [19]. The aim of this project is to define a feature construct for several languages,
and then to evaluate it in case studies. The approach bases on the superimposition idea of Katz [20], where we specify a base system first, and where we then specify textual increments by a suitable operator. For example Plath and Ryan [21] do this, they define a feature construct for CSP. They use a state-oriented specification style, and their feature operator allows to add or replace transitions at states. They don’t define type rules for their operator, instead they detect feature interactions by model-checking the combined systems. One problem of the superimposition idea is that, even though the textual increments have a defined interface, the base system does not. Therefore, a superimposition can invalidate arbitrary assumptions about the base system and thus cause feature interaction problems. We try to reduce these problems by encapsulating one self-contained property each in one section, and by either including or excluding it entirely.

Also related is the foreground/background approach of Hall [22]. He separates the description of a feature into its foreground behaviour, which is “essential” behaviour in our terms, and its background behaviour, which is “changeable” by another feature’s foreground behaviour without causing a warning by his analysis tools. His goal is to perform feature interaction detection with less results; i.e., his tools do not report “spurious” interactions. His approach does not provide multiple layers of features, as ours does, even though he mentions a generalization to an “n-ground” approach. We use a hierarchical, constraint-oriented structuring of features, and we are in particular concerned about avoiding feature interactions. Hall attempts to cope with current telephony system architectures, while we investigate a more suitable architecture.

7 SUMMARY

We are concerned with the maintenance of formal requirements documents. In the application area of telephone switching, so-called feature interaction problems have become a serious obstacle to evolve these systems according to market needs.

We propose a specification methodology that either 1) avoids feature interaction problems in requirements documents from the beginning, or 2) helps to detect them, such that they can be resolved. The original specifier of a feature must document additional information on what is the core of the feature, and what are the dependences on other features. Keeping all the variants and versions of a system as a single family enables us to exploit this information.

For specifying families, we present in this paper (and in full detail in the manual [12]) the syntax and semantics of a formal feature combination mechanism. It is an extension to the specification language CSP-OZ. In our formalism, some feature interactions appear as type errors.

In this paper, we also present a tool which checks for feature interaction problems, which extracts and transforms a desired family member from the family document, and which generates documentation on the structure of the family. The tool is available freely.

We finally report on a quite large case study. We show how a standard example of feature interactions does not occur due to the modular structure we use, and we show how our type checker detects a further feature interaction, which we then resolve.
Several issues are worth further research. More experience with our methodology is important; we plan to extend our case study further. Our approach should be applicable with other formalisms than CSP-OZ as well, as long as the base formalism used allows for a constraint-oriented specification style and provides a means for incremental refinement. Our tool even already supports the languages CSPZ and Z. Finally, the relationship between families of requirements and families of programs needs to be investigated, with respect to a reuse-oriented implementation process.

REFERENCES

Abstract  Subtyping relations for object-oriented formalisms describe relationships between super- and subclasses which satisfy the substitutability requirement imposed on types and their subtypes. Behavioural subtyping is concerned with subtypes for active classes with an explicit dynamic behaviour, specifiable for instance by object-oriented formal methods combining state-based with behavioural formalisms.

In this paper we develop syntactic patterns and semantic rules for the state-based part of a subclass which guarantee that the subclass is a behavioural subtype of its superclass. This allows to check for subtypes without computing the behavioural semantics of the class at all. Our results are similar to the ones linking data refinement in state-based methods with failure-divergence refinement in CSP. In contrast to classical data refinement, subtyping has to cope with additional new operations in the subclass.

Keywords:  Object-orientation, refinement, behavioural subtyping, Z, CSP.

1. INTRODUCTION

Inheritance is one of the key features for the success of object-oriented analysis and design methods as well as programming languages. Inheritance is employed to re-use already existing class descriptions and extend them with additional attributes and methods, thus creating a new subclass from an existing superclass. Subtyping for object-oriented formalisms [1, 6, 12] studies relationships between super- and subclasses with the aim of clarifying when a subclass can be said to be type substitutable for its superclass. Type substitutability means that “a subtype instance can be used wherever a supertype instance is expected” [17]. Subtyping is also studied in the context of active classes with a behavioural semantics (as for instance obtained when using LOTOS or CSP as a specification language). Active classes have an explicit
specification of the dynamic behaviour which describes synchronisation con-
straints of the class or certain protocols which have to be obeyed. In this paper,
we use the term behavioural subtyping to refer to subtyping for this type of
classes. A number of proposals for behavioural subtyping relations have been
made (among others [16, 2, 4, 13]; see conclusion for a discussion of related
work), with for instance Petri nets or labelled transition systems as semantic
basis. The starting point of this paper are three subtyping relations from [9]
with proven substitutability properties, two of them also adequate for shared
mutable objects. The formal basis for these two relations are transition systems
and their (CSP) failures and divergences.

Here we will be concerned with checking subtype relationships among classes
specified in object-oriented formal methods combining state-based with be-
havioural aspects. The formal method which we have in mind (and which we
use throughout our examples) is CSP-OZ [7], a combination of CSP
[10] and Object-Z [15], but the theory we develop will be independent of this
particular choice. Two aspects of CSP-OZ are however of importance: first,
the fact that the state-based part of CSP-OZ has a behavioural semantics (and
thus we have to use behavioural subtyping relations), and second, that the sem-
ants of the combination is obtained by separately computing the semantics
of the Object-Z and CSP part, and afterwards joining them by CSP parallel
composition. This second point allows us to look at state-based and behaviour
part in isolation when checking for subtypes.

A straightforward approach to check subtyping relationships among classes
is the following: given two class descriptions, compute their behavioural se-
manics and compare the classes according to some chosen relation. How-
ever, since the semantics of CSP-OZ (and most other such types of integrated
formal methods) has to be constructed by hand, this approach is in general
impracticable. Instead we are interested in developing simple, at the best syn-
tactic, patterns, which allow to avoid the explicit check and instead guaran-
tee subtypes by construction. Such patterns or rules have to give conditions
on attributes and operations in subclasses, which, when followed, immedi-
ately lead to subtypes. In this paper we will solely be interested in subtyp-
ing patterns for the state-based part (which, due to the compositional seman-
tics, can without change be combined with patterns for the behaviour part).
Our investigations are similar to the work about the relationship between data
refinement rules in state-based formalisms and failure-divergence refinement
[11, 18]. There, it has been studied whether data refinement induces failure-
divergence refinement when the state-based formalism is given a behavioural
semantics. The basic approach taken here is the same: we assume to have
a behavioural semantics for a state-based formalism, have a set of conditions
on two class specifications and show that the conditions induce a behavioural
subtyping relationship between the classes (soundness of rules).
We illustrate our approach by a small example of a buffer which is extended by means of inheritance in three different ways yielding three different kinds of subtypes.

2. CLASSES

We start with the specification of the buffer. We use CSP-OZ [7] syntax in the specifications. The following class specifications will not contain any CSP behaviour descriptions since we are only interested in patterns for the state-based part. Thus the classes almost look like Object-Z classes. The specified buffer can hold elements (in a finite bag) of a basic type \textit{Element} up to some arbitrary number \( n \). Elements may be put into the buffer and retrieved by means of the methods \textit{put} and \textit{get}, and an attribute \textit{alive} is used to indicate whether some specific instance of class \textit{Buffer} is alive or has already been deleted.

\begin{center}
\begin{Verbatim}
[Element]

\texttt{Buffer (n : } \mathbb{N} \texttt{)}

\begin{itemize}
  \item method \texttt{put : } [ \texttt{el? : Element} ]
  \item method \texttt{get : } [ \texttt{el! : Element} ]
\end{itemize}

\begin{itemize}
  \item \texttt{contents : } \texttt{fbag Element}
  \item \texttt{alive : } \texttt{B}
  \item \# contents \leq n
\end{itemize}

\begin{itemize}
  \item \texttt{enable-put}
    \texttt{el? : Element}
  \item \# contents < n \land alive
\end{itemize}

\begin{itemize}
  \item \texttt{effect-put}
    \texttt{\Delta (contents)}
    \texttt{el? : Element}
    \texttt{contents' = contents \cup \{el?\}}
\end{itemize}

\begin{itemize}
  \item \texttt{enable-get}
    \texttt{# contents \geq 1 \land alive}
\end{itemize}

\begin{itemize}
  \item \texttt{effect-get}
    \texttt{\Delta (contents)}
    \texttt{el! : Element}
    \begin{itemize}
      \item \texttt{el! \in contents}
      \item \texttt{contents' = contents \cup \{el!\}}
    \end{itemize}
\end{itemize}
\end{Verbatim}
\end{center}

The class is parameterised in the length of the buffer. The first two lines define the \textit{interface} of the class: it has methods \textit{put} and \textit{get}. Next the attributes and their initial values are defined. The application of methods \textit{put} and \textit{get} is described by enable and effect predicates, the enabling predicate specifying a \textit{guard} for the application of the method, the effect predicate the possible results of applying the method.
In general we represent a class \( C \) by a state schema, an initialisation schema and a number of operation schemas: \( C = (\text{State}, \text{Init}, (\text{enable}_j \text{Op})_{j \in J}, (\text{effect}_j \text{Op})_{j \in J}) \). The components have the following meaning: State describes the state space of the class. It is determined by giving a set of attributes \( \text{Attr} \) and their types together with an invariant \( \text{Inv} \) further restricting the allowed values. Init describes the set of initial states determined by a predicate over \( \text{Attr} \). \((\text{Op})_{j \in J}\) gives a family of methods or operations on the state space. Every operation \( \text{Op}_j \) has a number of input variables \((\text{in}_{j,i})_{1 \leq i \leq k_j}\) of type \( T_{\text{in}_{j,i}} \) and a number of output variables \((\text{out}_{j,i})_{1 \leq i \leq l_j}\) of type \( T_{\text{out}_{j,i}} \). Two schemas are given for every operation (an omitted schema in the CSP-OZ specification corresponds to \( \text{true} \)):

1. an enabling schema \( \text{enable}_j \text{Op} \) whose predicate ranges over the current state \( \text{Attr} \) and input variables \( \text{in}_{j,i} \). It acts as a guard to the operation execution, and

2. an effect schema \( \text{effect}_j \text{Op} \) with predicate ranging over current state \( \text{Attr} \), next state \( \text{Attr}' \), input variables \( \text{in}_{j,i} \) and output variables \( \text{out}_{j,i} \) which describes the postcondition of the operation (where \( \text{Attr}' \) refers to the new state of attributes after execution of the operation).

We use the following abbreviation for input and output variables:

\[
\begin{align*}
\text{In}_j & \equiv [\text{in}_{j,1} : T_{\text{in}_{j,1}}; \ldots ; \text{in}_{j,k_j} : T_{\text{in}_{j,k_j}}] \\
\text{Out}_j & \equiv [\text{out}_{j,1}! : T_{\text{out}_{j,1}}; \ldots ; \text{out}_{j,l_j}! : T_{\text{out}_{j,l_j}}]
\end{align*}
\]

Additionally we define the precondition of an operation \( \text{Op}_j \) to be \( \text{pre}_j \text{Op} : = \exists \text{State}', \text{Out}_j : \text{effect}_j \text{Op} \).

Next we provide the class description with a behavioural semantics in terms of labelled transition systems. The labels are taken from an alphabet \( \Sigma \) which contains all events \( \text{op}.i_1 \ldots .i_k .o_1 \ldots .o_l \) where \( \text{op} \) is the name of a method and the rest are possible values of input and output parameters. For an event \( \text{ev} = \text{op}.i_1 \ldots .i_k .o_1 \ldots .o_l \) we write \( \text{ch} (\text{ev}) = \text{op} \) to refer to the method or operation name of an event. \( \Sigma_{\tau} \) stands for \( \Sigma \cup \{ \tau \} \). A labelled transition system (LTS) is a tuple \( T = (Q, \rightarrow, Q^\text{in}) \) with \( Q \) a set of states, \( \rightarrow \) a transition relation and \( Q^\text{in} \) the set of initial states. A state \( q \) is said to be stable if it has no outgoing \( \tau \) transitions, \( q \) diverges \( (q \uparrow) \) if an infinite number of \( \tau \) steps is possible from \( q \), and \( \text{next}(q) \) is the set of all events which are executable from \( q \).

The basic idea of the semantics is to view a class as a process which is continuously executing enabled operations and thus changes its state. Initially it is in one of the states of \( \text{Init} \). The symbol \( \perp \) is used to indicate one particular divergent state: The class diverges if an enabled operation is applied outside its precondition. In the following we take a schema to stand for a set of bindings and for a binding \( q = \{ a_1 \Rightarrow v_1, \ldots , a_n \Rightarrow v_n \} \) and a schema \( S \) we write \( S(q) \).
if $q$ is a valid binding of schema $S$. The operator $\cup$ is used to join bindings: $\{ a_1 \Rightarrow v_1, \ldots, a_n \Rightarrow v_n \} \cup \{ b_1 \Rightarrow w_1, \ldots, b_m \Rightarrow w_m \} = \{ a_1 \Rightarrow v_1, \ldots, a_n \Rightarrow v_n, b_1 \Rightarrow w_1, \ldots, b_m \Rightarrow w_m \}$ assuming disjointness of the sets $\{ a_1, \ldots, a_n \}$ and $\{ b_1, \ldots, b_m \}$. For a state $q$, the notation $q'$ is used to describe the same binding but with components decorated with $'$, the notation $q|_{\text{State}}$ is a binding which only gives values to variables in $q$ which are declared in $\text{State}$.

**Definition 2.1**

The operational semantics of a class $C$ is defined as $[C] = (\mathcal{Q}_C, \rightarrow_C, \mathcal{Q}_C^{\text{in}})$, where

- $\mathcal{Q}_C = \text{State} \cup \{ \bot \}$, $\mathcal{Q}_C^{\text{in}} = \text{Init}$ and
- $\rightarrow_C$ is defined as $\bot \rightarrow_C \bot$ and
  
  - $q \xrightarrow{e_C} q_2$ if $ev = \text{op}_j \cdot i_1, \ldots, i_{k_j} \cdot \text{out}_1, \ldots, \text{out}_{l_j}$ and 
    
    $\text{enable}_{op_j}(q_1 \cup \{ \text{in}_n \Rightarrow i_n, 1 \leq n \leq k_j \})$ and
    
    $\text{effect}_{op_j}(q_1 \cup q_2 \cup \{ \text{in}_n \Rightarrow i_n, \text{out}_m \Rightarrow o_m, 1 \leq n \leq k_j, 1 \leq m \leq l_j \})$,

  - $q \xrightarrow{e_C} \bot$ if $ev = \text{op}_j \cdot i_1, \ldots, i_{k_j} \cdot \text{out}_1, \ldots, \text{out}_{l_j}$ and
    
    $\text{enable}_{op_j}(q_1 \cup \{ \text{in}_n \Rightarrow i_n, 1 \leq n \leq k_j \})$ but not $\text{pre}_{op_j}(q_1 \cup \{ \text{in}_n \Rightarrow i_n, 1 \leq n \leq k_j \})$.

Note that we use state $q_2'$ here since the effect predicate uses primed variables to refer to the state after execution.

The dynamic behaviour of class $\text{Buffer}(2)$ with an instantiation of type $\text{Element}$ with the set $\{1,2\}$ is depicted by the transition system in Figure 1.

Next we present three extensions of the basic class $\text{Buffer}$ by inheritance: the first adds a possibility of explicitly deleting the buffer, the second adds a new method checking for emptiness and the third a method which allows to insert two elements at once.

```plaintext
FinBuffer(n: N)

inherit Buffer(n)

method del

enable_del #contents = 0 ∧ alive

effect_del Δ( alive)

EmptyCheckBuffer (n: N)

inherit Buffer (n)
```
Conceptually inheritance is defined as conjunction on schemas: given an abstract class $A = (AState, AInit, (enable_{AOp_j})_{j \in J}, (effect_{AOp_j})_{j \in J})$ and specialised class $C = (CState, CInit, (enable_{COp_i})_{i \in I}, (effect_{COp_j})_{j \in J})$ assuming $C$ is constructed from $A$ by means of inheritance, then $J \subseteq I$ and

\[
\begin{align*}
CState &= AState \land ExtState \\
CInit &= AInit \land ExtInit,
\end{align*}
\]
\[
\begin{align*}
\text{enable}_{\text{COp}_j} &= \text{enable}_{\text{AOp}_j} \wedge \text{enable}_{\text{ExtOp}_j} \quad \forall j \in J, \\
\text{effect}_{\text{COp}_j} &= \text{effect}_{\text{AOp}_j} \wedge \text{effect}_{\text{ExtOp}_j} \quad \forall j \in J, \\
\text{enable}_{\text{COp}_i} &= \text{enable}_{\text{ExtOp}_i} \quad \forall i \in I \setminus J, \\
\text{effect}_{\text{COp}_i} &= \text{effect}_{\text{ExtOp}_i} \quad \forall i \in I \setminus J.
\end{align*}
\]

ExtState (ExtInit etc.) is the schema which is explicitly written in the subclass and AState (etc.) is inherited.

3. SUBTYPING

Subtyping on ordinary types is usually demanded to fulfill the requirement of type substitutability: subtypes should be usable wherever supertypes are expected without any noticeable differences [17]. When lifting this concept to classes, the first idea is to use the inheritance hierarchy for defining subtyping relations. This however does not fit the requirement of substitutivity anymore: inheritance is first of all concerned with re-use of class descriptions and allows fundamental changes to the behaviour of superclasses in a subclass. The requirement of substitutivity suggests that instead some sort of refinement should be used as a subtyping relation.

Since we are working in the context of CSP, our notion of refinement is failure-divergence refinement. In the process algebra CSP process refinement is used to compare an implementation against a specification with respect to their failures and divergences: the implementation has to be more deterministic than the specification, i.e. has less failures and divergences. Failures are sets of pairs \((\sigma, X) \in \Sigma^* \times 2^\Sigma\) with the meaning that a process has such a pair in its failure set if it can execute the trace \(\sigma\) and may thereby reach a stable state (no \(\tau\)-transitions possible) in which none of the events from \(X\) are possible. Divergences are sets of traces \(\sigma\) such that a process may diverge after the trace, i.e. execute an infinite number of invisible actions. Failures and divergences can for instance be computed from the transition system of a process, but due to lack of space we refrain from giving a formal definition here. Buffer(2) for instance has failures (with maximal refusals) \((\varepsilon, \{\text{get.1, get.2}\}), (\text{put.1, get.2}), (\text{put.1 \cap put.2, \{put.1, put.2\}}), \ldots\) and an empty set of divergences.

Refinement then simply is inclusion of failure and divergence sets: A labelled transition system \(I\) is a failure-divergence refinement of an Ts \((S \models F, D)\) iff \(\text{failures}(I) \subseteq \text{failures}(S)\) and \(\text{divergences}(I) \subseteq \text{divergences}(S)\).

According to this definition none of the subclasses of Buffer are refinements (e.g. FinBuffer has a failure \((\text{del, } \Sigma)\) which is not in Buffer’s failure set). Nevertheless refinement will be the basis for our three behavioural subtyping relations. The relations additionally have to cope with the fact that the subclass may introduce new methods. The comparison of sub- with superclass has to
"ignore" these methods or at least relate new methods to old ones. For this, we use three different operators: restriction, concealment (next) and substitution (below), and define them by their effect on failures and divergences.

**Definition 3.1** Let $T = (Q, \rightarrow, Q^m)$ be a labelled transition system and $N \subseteq \Sigma$.

- **Restriction**: $\text{failures}(T \setminus_r N) = \{ (\sigma, X) \mid \exists \ Y \bullet (\sigma, Y) \in \text{failures}(T), \sigma \in (\Sigma \setminus N)^*, X \subseteq Y \cup N \}$.

- **Concealment**: $\text{failures}(T \setminus_c N) = \{ (\sigma, X) \mid \exists (\sigma', Y) \in \text{failures}(T), \sigma = \sigma' \upharpoonright (\Sigma \setminus N) \land X \subseteq Y \cup N \}$.

Divergences are unchanged by restriction and concealment.

Restriction forbids the execution of events in $N$, concealment makes $N$-events invisible. Note that concealment differs from classical hiding in process algebras. Hiding may usually introduce new non-stable states, whereas concealment does not.

The basic idea of substitution is to map new methods onto sequences of old methods: the state changes achieved by new method should be the same as that achieved by a consecutive execution of the old methods. This is the “dynamic behaviour” counterpart of Liskov and Wing’s extension maps [12].

**Definition 3.2** Let $(\sigma, X) \subseteq \Sigma \times 2^\Sigma$ be a pair of trace and refusal, and $N \subseteq \Sigma$ a set of events (the new events to be mapped onto old ones). Furthermore let $f_N : N \rightarrow \Sigma^*$ be a function, the substitution function. We define $f_N(\sigma, X)$ (by overloading the function $f_N$) to be $(f_N(\sigma), X \cup N)$ where $f_N(\sigma)$ is inductively defined by $f_N(\varepsilon) = \varepsilon$ and

$$f_N(a \triangleleft \sigma) = \begin{cases} a \triangleleft f_N(\sigma) & \text{if } a \notin N \\ f_N(a) \triangleleft f_N(\sigma) & \text{else} \end{cases}$$

Superclass and subclass are then compared by applying concealment, restriction or substitution to the new methods before checking for failure and divergence inclusion (see [9]).

**Definition 3.3** Let $A$, $C$ be labelled transition systems, $A$ standing for the super- and $C$ for the subclass, and let $N$ be the set of events over new methods.

- $C$ is a weak subtype of $A$ (denoted $A \subseteq_{\text{wst}}^N C$) iff $A \subseteq_{\text{iTD}} C \setminus_r N$.
- $C$ is a safe subtype of $A$ according to a substitution function $f_N$ (denoted $A \subseteq_{\text{sst}}^N C$) iff $\forall (\sigma, X) \in \text{failures}(C) : f_N(\sigma, X) \in \text{failures}(A)$ and $\forall \sigma \in \text{divergences}(C) : f_N(\sigma) \in \text{divergences}(A)$.
- $C$ is an optimal subtype of $A$ (denoted $A \subseteq_{\text{ost}}^N C$) iff $A \subseteq_{\text{iTD}} C \setminus_c N$. 
Coming back to our four buffers: *FinBuffer* is a weak subtype of *Buffer* and *EmptyCheckBuffer* is a weak as well as an optimal subtype. *FinBuffer* is not an optimal subtype of *Buffer* since \( \mathcal{E} \) has a failure \(( \varepsilon, \Sigma )\) which *Buffer* does not have. Finally *2Buffer* is a safe subtype (mapping \( \text{put} 2\) to \( \text{put} \) \( \cup \) \( \text{put} \)) as well as a weak subtype of *Buffer* but no optimal subtype.

In order to check for subtypes we first had to construct the transition systems, compute failures/divergences for the classes and afterwards apply the subtyping definition. Next we look at *patterns* which give us the possibility of obtaining a subtype by *construction* and *rules* which allow to *locally* check for subtypes without computing the semantics of classes at all.

### 4. PATTERNS AND RULES

We refer to *patterns* as conditions on the specification of a subclass which can be checked (almost) completely syntactically, without looking at the semantics at all. *Rules* will be conditions on classes which can be checked by looking at the semantics of methods in isolation, i.e. without computing the transition system semantics of a class but by comparing the meaning of methods of sub- and superclass.

Besides the patterns and rules discussed here, it is always possible to employ *data refinement rules* (specific to CSP-OZ [8]): any subclass which is a refinement of its superclass is a subtype in the sense of all three relations. All of the following patterns and rules can thus be combined with a valid data refinement.

**Patterns.** We present two patterns here, one for weak and one for optimal subtyping. The patterns are conditions on the changes and extensions allowed in the subclass. We again assume \( A = (A\text{State}, A\text{Init}, (AOp_j)_{j \in I}) \) to be the super- and, letting \( J \subseteq I \), \( C = (C\text{State}, C\text{Init}, (COp_j)_{j \in I}) \) to be the subclass.

**P1** \( \forall j \in J : \text{enable}_COP_j = \text{enable}_AOP_j \) and \( \text{effect}_COP_j = \text{effect}_AOP_j \), i.e. the specialised subclass does not change the old methods at all. There are no restrictions on new methods or attributes.

**P2**

**P2.A** \( \forall j \in J : \text{enable}_COP_j = \text{enable}_AOP_j \) and \( \text{effect}_COP_j = \text{effect}_AOP_j \),

**P2.B** \( \forall n \in I \setminus J : \) the predicate of schema \( \text{effect}_COP_n \) ranges solely over \( \text{Attr}, \text{ExtAttr} \) and \( \text{ExtAttr'} \), i.e. no changes to old attributes are allowed,

**P2.C** \( \forall n \in I \setminus J : \text{enable}_COP_n \Rightarrow \text{pre}_COP_n \).

Here = stands for syntactical equality. Pattern P1 is a pattern for a weak subtype, e.g. *FinBuffer* may be constructed from *Buffer* using pattern P1, it leaves
all old methods unchanged. Pattern P2 is valid for EmptyCheckBuffer: the new method empty does not change old attributes and its precondition holds whenever the method is enabled. We however cannot construct FinBuffer from Buffer using pattern P2 since FinBuffer’s effect predicate for operation del changes a value of an old attribute (alive). Condition P2.C is essentially used to avoid new divergences in the subclass. It is the only part of the patterns which is not purely syntactic, but this is necessary since we want to take divergence into account.

The following main theorem now states the soundness of the patterns for the subtyping relations.

**Theorem 1**

If C is a subclass of A obeying pattern P1 then $A \sqsubseteq_{wst} C$.

If C is a subclass of A obeying pattern P2 then $A \sqsubseteq_{ost} C$.

Note that such a relationship cannot be guaranteed by inheritance alone. Inheritance allows all kinds of changes in the subclass. Due to lack of space we only show the proof for the second part of the theorem. For the proof we show that, when constructed by a pattern, subclass and superclass are even more close to each other than defined by subtyping: the superclass simulates the subclass in a certain sense. Hence in the following we define a simulation relation inducing optimal subtyping. The relation is in spirit a forward simulation. We do not consider backward simulations here since we are not interested in completeness. In fact, it is immediately clear that the patterns are not complete: there are as well valid subtypes which cannot be constructed according to the patterns.

For the simulation definition we assume that the semantics of Definition 2.1 is used for classes, with the consequence that there are no $\tau$-transitions except from $\bot$ to $\bot$.

**Definition 4.1** Class $A$ simulates $C$ concealing $N$, $A \preceq_{(N)} C$, if there is a relation $\rho \subseteq Q_A \times Q_C$ such that

1. $\forall q_C \in Q_C^{in} \exists q_A \in Q_A^{in} : (q_A, q_C) \in \rho$,  
2. $\forall (q_A, q_C) \in \rho$ the following holds:
   - $\text{stable}(q_C) \Rightarrow \text{stable}(q_A)$, $\text{next}(q_C) \setminus N = \text{next}(q_A)$, $q_C \uparrow \Rightarrow q_A \uparrow$,
   - $\forall a \not\in N : q_C \xrightarrow{a} q_C' \Rightarrow \exists q_A' : q_A \xrightarrow{a} q_A' \wedge (q_A', q_C') \in \rho$,
   - $\forall a \in N : q_C \xrightarrow{a} q_C' \Rightarrow (q_A, q_C') \in \rho$.

The idea behind simulation for optimal subtypes is that for the $N$-steps of the subclass we do not require a matching step of the superclass, instead states $q'$
reached after executing new methods from some state $q$ must be simulated by the matching partner of $q$.

**Theorem 2** $A \preceq (N) C \Rightarrow A \subseteq_{ost}^N C$.

The remaining part for proving (the second part of) Theorem 1 is to show that subclasses constructed by patterns can be simulated by the superclasses. We again use $N$ to refer to the set of all events over new methods.

**Theorem 3** If $C$ is a subclass of $A$ obeying pattern $P_2$ then $A \preceq (N) C$.

**Rules.** We proceed with giving rules for the last subtyping relation, safe subtyping. Here, we actually have to compare the semantics of new methods with that of (sequences of) old methods. This requires a little more technical overhead. We explain this by means of $2\text{Buffer}$. $2\text{Buffer}$ is a safe subtype of $\text{Buffer}$ since the new method $\text{put}^2$ can be explained in terms of the old method $\text{put}$: the effect of $\text{put}^2$ is the same as that of two $\text{put}$’s. However, the substitution function $f_N$ that we need does not map a new method/operation to a sequence of operations but events to sequences of events. Besides transformation of operations we also need some means for transforming inputs and outputs of a new operation onto the inputs and outputs of the old operations. For this, we use (an adaptation of) the technique of input and output transformers as for instance advocated in [5]. An input or output transformer simply is a schema that relates inputs (or outputs) of operations. Looking at $2\text{Buffer}$, the input transformer should relate $el^1?$ and $el^2?$ with the two inputs $el^1$? of two $\text{put}$’s. However, the output transformer is empty. In order to be able to distinguish the inputs of the two $\text{put}$’s we define another decoration on schemas, that decorate input and output variables with numbers.

For an operation schema $S = \{a_1 : T_1; \ldots; a_n : T_n, in^? : T_{in}, out^! : T_{out} \mid p\}$ we define $S^k, k \in \mathbb{N}$, to be $\{a_1 : T_1; \ldots; a_n : T_n, in^k? : T_{in}, out^k! : T_{out} \mid p[\text{in}^?/\text{in}^k?, \text{out}!/\text{out}^k!]\}$ (and similar for more than one input or output variable). The decoration is used to distinguish variables with the same name coming from operations at different positions in the substituting sequence. The input transformer from $\text{put}^2$ to $\text{put}^1 × \text{put}^2$ can thus be formulated as:

\[
\begin{array}{l}
\text{IT} \\
e 1? : \text{Elwmnt}; \text{el} 2? : \text{Element} \\
e 1? : \text{Element}; \text{el} 2? : \text{Element} \\
\text{e} 1? = \text{e} 1? \land \text{e} 2? = \text{e} 2?
\end{array}
\]

It transforms inputs of $\text{put}^2$ into outputs which can than be fed into $\text{put}^1 × \text{put}^2$ (using the Z pipe operator $\gg$). Conversely, an output transformer from a sequence of old operations $AOp_1 × AOp_2$ to a new operation $COp$ would take the
outputs of $AOp_1 \bowtie AOp_2$ as inputs and transform them into outputs of $COp$. The output transformer from \textit{put}$^1 \bowtie \textit{put}$ to \textit{put} is an empty schema: no declarations and the predicate is true.

To simplify matters, we only consider substitutions which map one operation to a sequence of two operations in the sequel. This can however easily be generalised to an arbitrary number of operations.

\textbf{Definition 4.2} An input transformer from $COp$ to $AOp_1 \bowtie AOp_2$ is a schema $IT$ s.t.

- all input variables from $COp$ are declared in $IT$,
- for every input variable $x^i$ of type $T$ from operation $AOp_i$, $i = 1, 2$, there is one output variable $x^i! : T$ in $IT$,
- no other variables are declared in $IT$ and
- the predicate of $IT$ is a conjunction of equalities of the form $x^? = y!$ such that every output variable occurs exactly once.

Conversely, an output transformer from $AOp_1 \bowtie AOp_2$ to $COp$ is a schema $OT$ in which all output variables from $COp$ are declared, for every output variable $x^i! : T$ from $AOp_i$, $i = 1, 2$, there is one input variable $x^i? : T$, no other variables are declared in $OT$, and the predicate of $IT$ is a conjunction of equalities of the form $x^? = y!$ such that every output variable occurs exactly once.

Figure 2 illustrates the use of input and output transformers: execution of $COp$ should be the same as that of $AOp_1 \bowtie AOp_2$, and the input and output transformers define the valid relationships between the input and outputs.

We now again assume $A$ to be the super- and $C$ to be the subclass. The following rule formalises Liskov and Wings’s extension maps [12] in our setting.

\textbf{Extension Rule:} The extension rule $R$ holds between $A$ and $C$ if all old methods remain unchanged in the subclass (or are changed according to data refinement rules) and there is an extension map $E : N_{op} \rightarrow O_{op}$ such that the following holds: for every $COp \in N_{op}$, either $COp$ does not change attributes at all and $E(COp) = \varepsilon$, or else $COp$ does change attribute values, $E(COp) = AOp_1 \bowtie AOp_2$ and there is an input transformer $IT$ from $COp$ to $AOp_1 \bowtie AOp_2$, an output transformer $OT$ from $AOp_1 \bowtie AOp_2$ to $COp$ and the following conditions are furthermore met.

\begin{align*}
R_A & \forall CState, \forall In \bullet \text{enable}_COp \Rightarrow IT \triangleright \triangleleft (\text{enable}_AOp_1 \land \text{pre}_AOp_1), \\
R_B & \forall CState, \forall In \bullet \text{enable}_COp \Rightarrow \text{pre}_COp, \\
R_C & \forall CState, CState', \forall In \bullet \text{enable}_COp \land (IT \triangleright \triangleleft \text{effect}_AOp_1) \\
& \Rightarrow IT \triangleright \triangleleft (\text{enable}_AOp_2 \land \text{pre}_AOp_2),
\end{align*}
The conditions have the following meaning: R.A states that whenever \( COp \) is enabled so is \( AOp_1 \) and its precondition is also fulfilled; R.B guarantees that \( COp \) does not introduce new divergences; R.C states that after execution of \( AOp_1 \) the second operation of the sequence \( AOp_2 \) is enabled and defined. Finally R.D guarantees that the effect of executing \( COp \) is the same as that of \( AOp_1 \circ AOp_2 \).

An extension map and the corresponding input and output transformers \( IT, OT \) induce a substitution function \( f_E^{IT,OT} : N \rightarrow \Sigma^* \). This function has to transform events into sequences of events; the extension map tells us which operation to map onto which sequence of operations and the input and output transformers tell us the values of parameters. Additionally we need the ordering among inputs and outputs as defined in the interface declaration of the class. Let \( COp \) and \( AOp_1, AOp_2 \) be defined by

\[
\text{method } COp : [cin_1? : T_1, \ldots, cin_k : T_k, cout_1 : T_{k+1}, \ldots, cout_l : T_{k+l}] \\
\text{method } AOp_1 : [ain_{1,1}? : T_{1,1}, \ldots, ain_{1,k_1} : T_{1,k_1}, aout_{1,1 : T_{1,k_1+1}, \ldots, aout_{1,l_1} : T_{1,k_1+l_1}] \\
\text{method } AOp_2 : [ain_{2,1}? : T_{2,1}, \ldots, ain_{2,k_2} : T_{2,k_2}, aout_{2,1 : T_{2,k_2+1}, \ldots, aout_{2,l_2} : T_{2,k_2+l_2}]
\]

Given this declaration we associate with an input transformer \( IT \) two functions \( \mathcal{I}^i \), \( i = 1, 2 \), mapping input values of \( COp \) onto input values of \( AOp_1, AOp_2 \).
respectively.

\[ \mathcal{IT}^t(i_1, \ldots, i_k) = (w_1, \ldots, w_k) \]

with \( w_j = i_m, 1 \leq j \leq k_i, 1 \leq m \leq k \), if \( \text{cin}_m = \text{ain}_m^j \) is an equation in \( \mathcal{IT} \). For output transformers \( \mathcal{OT} \) we have a single function that maps ordered outputs of \( \mathcal{AOp}_1 \) to outputs of \( \mathcal{COp} \).

\[ \mathcal{OT}(v_{1,1}, \ldots, v_{1,l_1}, v_{2,1}, \ldots, v_{2,l_2}) = (o_1, \ldots, o_l) \]

with \( o_j = v_{i,m} \) if \( \text{cout}_m = \text{aout}_m^j \) is an equation in \( \mathcal{OT} \). Then \( f^E_{\mathcal{IT}, \mathcal{OT}} \) is defined to be

\[ f^E_{\mathcal{IT}, \mathcal{OT}}(o.p.i_1, \ldots, i_k, o_{1}, \ldots, o_l) = E^1(op).\mathcal{IT}^t(i_1, \ldots, i_k).v_{1,1}, \ldots, v_{1,l_1} \rhd E^2(op).\mathcal{IT}^t(i_1, \ldots, i_k).v_{1,1+1}, \ldots, v_{l+1,l+2} \]

such that \( \mathcal{OT}(v_{1,1}, \ldots, v_{1,l_1}, v_{1,l_1+1}, \ldots, v_{l+1,l+2}) = (o_1, \ldots, o_l) \).

Looking at 2Buffer we find that \( E \) defined by \( E(\text{put}2) = \text{put} \rhd \text{put} \) is an extension map; for instance \#contents < n – 1 implies \#contents < n.

The following Lemma is the key to correctness of the extension rule:

**Lemma 4.1**

Let \( A = \langle \text{AState}, \text{AInit}, (\text{enable}_\mathcal{AOp})_{j \in J}, (\text{effect}_\mathcal{AOp})_{j \in J} \rangle \) be the super-class and \( C = \langle \mathcal{CState}, \text{CInit}, (\text{enable}_\mathcal{COp})_{j \in J}, (\text{effect}_\mathcal{COp})_{j \in J} \rangle \) the subclass. Let \( E : N_{\text{op}} \rightarrow O^* \) be an extension map with input and output transformers \( \mathcal{IT}, \mathcal{OT} \) for every new operation. Let \( T_c = \llbracket C \rrbracket \) and \( T_A = \llbracket A \rrbracket \) be the transition systems. Then the following holds:

\[ \forall q_1 \in Q_C : q_1 \xrightarrow{e^r} q_2 \text{ with } c(h(e)) \not\in N_{\text{op}} \text{ a new operation} \]

\[ \Rightarrow \exists q_3 \in Q_A : q_1 \xrightarrow{\text{AState} - e^v_1} q_3 \xrightarrow{e^v_2} q_2 \text{ such that } f^E_{\mathcal{IT}, \mathcal{OT}}(e^v_1) = e^r \rhd e^v_2. \]

The proof is again omitted. Analogous to the other subtyping relations, we have a simulation relation which induces safe subtyping.

**Definition 4.3** Let \( f_N : N \rightarrow \Sigma^* \) be a substitution function. Class \( A \) simulates \( C \) changing \( N \) by \( f_N \), \( A \subseteq f_N \) \( C \), if there is a relation \( \rho \subseteq Q_A \times Q_C \) such that

1. \( \forall q_C \in Q_C^\text{in} \exists q_A \in Q_A^\text{in} : (q_A, q_C) \in \rho \),
2. \( \forall (q_A, q_C) \in \rho \) the following holds:
   - stable\( (q_C) \Rightarrow \) stable\( (q_A), \) next\( (q_C) \setminus N = \) next\( (q_A), q_C \uparrow \Rightarrow q_A \uparrow \),
   - \( \forall a \not\in N : q_C \xrightarrow{a} q_C \Rightarrow \exists q'_A : q_A \xrightarrow{a} q'_A \land (q'_A, q'_C) \in \rho \),
   - \( \forall a \in N : q_C \xrightarrow{a} q_C \Rightarrow \exists q'_A : q_A \xrightarrow{f(a)} q'_A \land (q'_A, q'_C) \in \rho . \)
As before we get two results relating simulation with subtyping and patterns with simulation.

**Theorem 4** \( A \preceq_{fn} C \Rightarrow A \subseteq_{sst} C. \)

If \( C \) is a subclass of \( A \) obeying the extension rule \( R \) then \( A \preceq_{fn} C. \)

5. **DISCUSSION**

In this paper we presented subtyping patterns and rules for active classes with a behavioural semantics. Patterns are simple, mostly syntactic conditions which, when applied during construction of subclasses via inheritance, guarantee correct subtypes. Rules are conditions which can be locally checked on the semantics of operations.

**Related work.** First of all there is a huge amount of work on behavioural subtyping relations for object-oriented formalisms. They differ in the semantic models used as a basis and in the defined relations. A rough classification could divide them into bisimulation and refinement-based approaches. Bisimulation based approaches easily achieve some form of substitutivity since they use an equivalence relation for subtyping. We consider this to be too strong since subtyping in its basic nature is not symmetric. Refinement-based proposals on the other hand in most cases only discuss substitutability but do not give a formal proof. The relation most often used in this context is extension, developed for the process algebra LOTOS. Extension allows extension of functionality, but not only on new methods but also on old methods.

The discussion of related work concerning the specific topic of this paper can be split in two parts. The first one concerns the large amount of work on simulation relations for refinement, especially those showing the tight connections between data refinement in state-based and process refinement in behaviour-based systems [11, 18]. The ideas of these papers are clearly the basis for our simulation relations which adapt this basic concept to the case of behavioural subtyping relations. The second area are patterns for subtyping relations. The work of van der Alst and Basten [16] presents transformation rules on Petri nets which construct subtypes, where the subtyping relations are based on branching bisimilarity. Rudkin [14] studies subtyping patterns for LOTOS, i.e. specific LOTOS operators that induce subtypes. The relation chosen there is again extension.

Patterns which are similar to ours are also employed in UNITY’s superposition operator [3]. This is not an object-oriented setting, nevertheless the goal is similar: extend a program in a conservative way, not disturbing any of the properties holding beforehand. In fact, the idea of conservative extension appears in many theories and behavioural subtyping is just a form of conservative extension in object-orientation.
REFERENCES


Part Eight

Theories of Verification
This page intentionally left blank.
VERIFICATION OF DENSE TIME PROPERTIES USING THEORIES OF UNTIMED PROCESS ALGEBRA

Matti Luukkainen
University of Helsinki, Department of Computer Science
PO Box 26, FIN-00014 University of Helsinki, FINLAND
mluukkai@cs.helsinki.fi

Abstract The article shows how untimed process algebraic methods can be used in verifying a category of dense time properties from a restricted class of timed automata. The work is based on the known results that if we restrict ourselves to a certain class of timed automata, then most of the interesting dense time properties are verifiable using discrete time methods. Those results are refined here in order to fit the process algebraic context, and furthermore, reduction strategies based on behavior equivalences, compositionality and abstraction are indicated. The usability of the method is evaluated with two case studies.

Keywords: Formal verification, Timed automata, Process algebra, Compositionality

1. INTRODUCTION

Because of the state explosion problem, a useful principle in algorithmic verification is that the information unnecessary for a verification task should be abstracted away. Especially, we would like to see in the context of systems specified with timed automata when it is sufficient to use discrete time algorithms but still say something about the system behavior in dense time.

In Henzinger et al., 1992 it was shown that for systems specified with timed transition systems, a large class of dense time properties can be verified by using discrete time algorithms. These results were generalized to timed automata in Bošnački, 1999. In this article we show how the methods used in standard untimed process algebra, and the related tools, can be used in verifying this category of dense time properties
from the timed automata descriptions. We have refined the previously known results in order to fit the process algebraic context, and furthermore, some efficient reduction strategies based on behavior equivalences, abstraction and compositionality are advocated.

2. TIMED AUTOMATA

Informally, a timed automaton (Alur and Dill, 1994) is just a transition system extended with a set of a special type of variables, namely clocks which are used to control the executability and urgency of transitions with respect to real time.

Let $X$ be a finite set of non-negative valued variables, namely clocks. Let $\Phi$ be a set of predicates, defined as the smallest set satisfying:

$$\phi ::= true \mid x \sim c \mid \phi \land \phi,$$

where $x \in X$, $c \in \mathbb{N}$ and $\sim \in \{<,\leq,=,\geq,>\}$.

A timed automaton is a 6-tuple $(S, A, X, R, I, s_0)$, where

- $S$ is a finite set of control states,
- $A$ is a finite set of transition labels or actions,
- $X \subseteq X$ is a finite set of real-valued clocks,
- $R \subseteq S \times A \times \Phi \times 2^X \times S$ is the transition relation,
- $I : S \rightarrow \Phi$ is a function that labels control states with time progress conditions,
- $s_0 \in S$ is the initial control state.

In a transition $(s, a, \phi, Y, s') \in R$, $s$ is the source and $s'$ the destination control state, $\phi \in \Phi$ is the guard of the transition, and $Y \subseteq X$ denotes the set of clocks that are reset to zero during the transition.

A valuation $v \in V$ is a function that assigns a non-negative real value to each of the clocks, i.e. for every $x \in X : v(x) \in \mathbb{R}^\geq 0$. If $X$ denotes a set of clocks, valuation $v[X := 0]$ is defined as follows: if $x \in X$ then $v[X := 0](x) = 0$, else $v[X := 0](x) = v(x)$. If $t \in \mathbb{R}^\geq 0$, and $v$ is a valuation, then $v + t$ is a valuation such that $(v + t)(x) = v(x) + t$ for all the clocks $x$.

A state of a timed automaton is its control state and a valuation of the clocks. Let $Q \in S \times V$ be the set of states of a timed automaton. A timed automaton can evolve in two ways, either executing a discrete transition or letting time pass without changing the control state. Possible transitions are formulated by the two rules below.

- Let $(s, a, \phi, Y, s')$ be a transition. The state $(s, v)$ has a discrete transition labeled with $a$ to $(s', v')$, if $\phi(v)$ evaluates to true and $v' = v[Y := 0]$.
- Let $t \in \mathbb{R}^\geq 0$. The state $(s, v)$ has a time transition labeled with $t$ to $(s, v + t)$ if for all $t' \leq t$ the time progress condition $I(s)(v + t')$ is true.

As can be noticed, the state space of a timed automaton can be defined with a transition system. Its alphabet consists of the transition
labels of the timed automaton (the discrete transitions) and positive real values (the time transitions). Formally, given a timed automaton $T\!A = (S, A, X, R, I, s_0)$, its state space is defined by a transition system $S_{T\!A} = (S \times \mathcal{V}, A \cup \mathbb{R}_{\geq 0}, R^t, (s_0, v_0))$, where $R^t$ is the smallest set containing all discrete and timed transitions of the timed automaton (which are given by the two rules above), and $v_0$ is a valuation that assigns the value zero to all the clocks.

The most natural way of modeling a system is to describe each of the components with a separate timed automaton. The behavior of the entire system is then defined by the timed automaton resulting from the parallel composition (denoted with $\parallel$) of the component automata.

Now we will give an automata theoretic semantics of timed automata. A timed word is a pair $(\vec{\sigma}, \vec{t})$, where $\vec{\sigma} = \sigma_1, \sigma_2 \ldots$ is an infinite sequence of transition labels $\sigma_i \in A$, and $\vec{t} = t_1, t_2 \ldots$ is an infinite, monotone and progressive sequence of time values. The intuition behind the timed word is that a transition labeled with $\sigma_i$ has occurred at time $t_i$ from the beginning of the observation. All the possible timed words with time domain $\mathbb{T}$ are denoted $TW_\mathbb{T}$.

If we add to the definition of timed automaton a set of accepting states $F \subseteq S$, then it can be thought of as a timed Büchi-automaton which accepts a set of timed words.

An execution of timed automaton $T\!A$ which corresponds to the timed word $(\vec{\sigma}, \vec{t})$ is an infinite sequence

$q = q_1 t_1 \rightarrow q'_1 \xrightarrow{\sigma_1} q_2 t_2 - t_1 \rightarrow q'_2 \xrightarrow{\sigma_2} q_3 t_3 - t_2 \rightarrow q'_3 \xrightarrow{\sigma_3} q_4 \ldots$,

in the $S_{T\!A}$, so each $q_i, q'_i$ are of the form $(s, v)$. The intuition behind the execution is that at the $i$th step, the automaton first lets time progress for $t_i - t_{i-1}$ units and then executes a transition labeled with $\sigma_i$.

A timed automaton $T\!A$ accepts the timed word $(\vec{\sigma}, \vec{t})$ if it has an execution $q$ corresponding to that word, starting from the initial state $(s_0, v_0)$ and which contains infinite occurrences of states $(s_i, v_i)$ such that $s_i \in F$. Language accepted by a timed automaton $T\!A$ is defined as usual:

$[T\!A] = \{(\vec{\sigma}, \vec{t}) | T\!A \text{ has an accepting computation corresponding to the word } (\vec{\sigma}, \vec{t}) \}$.

In the automata theoretic setting, the question whether a system $S$ satisfies a requirement $\varphi$ (denoted $S \models \varphi$) can be formulated as follows. The system and the property are defined as two timed Büchi-automata $T\!A_S$ and $T\!A_\varphi$. The automata define two sets of timed words $[T\!A_S]$ and $[T\!A_\varphi]$. Now the question whether $S \models \varphi$ is equivalent to the inclusion $[T\!A_S] \subseteq [T\!A_\varphi]$. The inclusion can be replaced by the emptiness check of $[T\!A_S] \cap [T\!A_{\neg \varphi}]$, where $T\!A_{\neg \varphi}$ is a timed Büchi-automata that defines the negation of the property $\varphi$. 
Intersection \([TA_1] \cap [TA_2]\) of the languages accepted by two timed Büchi-automata equals the language accepted by the product automaton \([TA_1 \times TA_2]\) (Alur and Dill, 1994). In the special case where all the states of the other automata, e.g. \(TA_1\) are accepting, the product automaton \(TA_1 \times TA_2\) corresponds to the parallel composition \(TA_1 \parallel TA_2\), where the accepting states are those in which the component corresponding to \(TA_2\) is in accepting state. Since the automaton \(TA_S\) that describes the system usually has all the states as accepting ones, the verification can be done by checking whether \([TA_S \parallel TA_\varphi] = \emptyset\).

3. VERIFICATION

It is known that several interesting verification problems are decidable for timed automata. Essentially the method of showing the decidability, and also the way of conducting the verification is the construction of a region graph, which is a finite abstraction of the state space of a timed automaton.

The dense time verification methods based on finite abstractions are computationally quite hard. Furthermore, the decidability of the dense time methods does not allow the clock values to be compared against arbitrary real values, but instead just integers. A question that now arises is, whether we could use discrete time verification algorithms in deducing dense time properties.

In Henzinger et al., 1992 it was shown that for systems specified with timed transition systems, a large class of dense time properties can be verified by using discrete time algorithms. These results were generalized to timed automata in Bošnaki, 1999.

We have taken these known results as our starting point in developing our verification methodology. Since the theory in Henzinger et al., 1992 and Bošnaki, 1999 is quite involved, we just rephrase here the most interesting consequences from our point of view. See the full version of this paper (M. Luukkainen, 2001) for full treatment and motivation for the result.

Let \(\Pi\) be a set of timed words. Let us denote with a subset of timed words of \(\Pi\), which consists only of those words where the observation times \(t_i\) are natural numbers, formally \(Z(\Pi) = \Pi \cap TW_\mathbb{N}\). In other words, if \([TA]\) characterizes the behavior of an automaton \(TA\) within dense time semantics, \(Z(TA)\) would be its interpretation in the discrete time model. Now the question that we are interested in is, what can be concluded about the dense time behavior of an automaton \(TA\) by verifying \(Z(TA) \subseteq Z(TA_\varphi)\), or \(Z(TA) \cap Z(TA_\varphi) = \emptyset\), for a property \(\varphi\). Thus,
we want to know what dense time properties of systems can be deduced by using discrete time methods.

Firstly, let us call a timed automaton where relations only of the form \(\leq, =, \geq\) are used as a weakly constrained timed automaton. Analogously let us call a timed automaton where only \(<, >\) or the trivial constraint \textit{true} are used as a strongly constrained timed automaton.

Now the following theorem (Henzinger et al., 1992; Bošnački, 1999) describes the relation between discrete and dense time verification.

\textbf{Theorem 1} Let \(TA_1\) and \(TA_2\) be two weakly constrained and \(TA_3\) a strongly constrained timed Büchi automata.

1. \([TA_1] \subseteq [TA_3]\) if and only if \(Z(TA_1) \subseteq Z(TA_3)\).
2. \([TA_1] \cap [TA_2] = \emptyset\) if and only if \(Z(TA_1) \cap Z(TA_2) = \emptyset\).
3. \(Z(TA_1) \cap Z(TA_2) = Z(TA_1 \parallel TA_2)\).

The above theorem guarantees the following: if a system is described with weakly constrained timed automaton \(TA_S\) and the negation of the correctness requirements is given as another weakly constrained automaton, say \(\overline{\varphi}\), the verification task whether \(S \models \varphi\) can now be reduced to checking whether the language accepted by the parallel composition, of \(TA_S\) and \(TA_{\overline{\varphi}}\) is empty in the discrete time domain. Since the result of the parallel composition itself is just one timed automaton (with a set of accepting states), we can now concentrate on checking the language emptiness of a single timed automaton in the discrete time interpretation.

As stated in the previous section, a timed automaton \(TA\) accepts a timed word \((\sigma, \bar{t})\) if and only if the corresponding state space \(S_{TA}\) has a path which corresponds to the timed word, and the path visits an accepting state infinitely often. So, in order to check the emptiness of \(TA\) one could observe whether the condition holds in the transition system \(S_{TA}\). But since \(S_{TA}\) is inherently an infinite structure, even in the discrete time interpretation that we now only consider, it is not practical for verification purposes.

In the discrete time domain, the reason for infiniteness is the fact that clock values can potentially increase without bounds. In every timed automaton there exists the greatest value \(x_c\) against which a clock \(x \in X\) is compared. When the value of \(x\) is greater than \(x_c\), the further increase is no more meaningful to the behavior of the system. So, keeping this in mind, we can modify the rules for transitions of timed automata in the following way (the first rule is unaltered):

- Let \((s, a, \phi, Y, s')\) be a transition. The state \((s, v)\) has a discrete transition labeled with \(a\) to \((s', v')\), if \(\phi(v)\) evaluates to true and \(v' = v[Y := 0]\).
- Let \(t \in \mathbb{R}^\geq 0\). The state \((s, v)\) has a time transition labeled with \(t\) to \((s, v')\).
where \( v'(x) = v(x) + t \) if \( x \leq x_c \), else \( v'(x) = x_c \),

if for all \( t' \leq t \) the time progress condition \( I(s)(v + t') \) is true.

Now we can define the *reduced state space* \( RS_{TA} \), a finite structure that can be used to derive the information about the executions of the timed automaton \( TA \) in the discrete time interpretation.

**Definition 2** Given a timed automaton \( TA = (S, A, X, R, I, s_0) \), its reduced state space is defined by a transition system \( RS_{TA} = (S \times \mathbb{V}, A \cup \mathbb{R}^{\geq 0}, R^t, (s_0, v_0)) \), where \( R^t \) is the smallest set containing all discrete and timed transitions (given by the two rules above) of the timed automaton in the discrete time domain.

The next theorem summarizes two important properties of transition systems corresponding to reduced state spaces. The proofs can be found in the full version of the paper (M. Luukkainen, 2001).

**Theorem 3** Let \( TA \) and \( TA' \) be two timed automata.

1. There is a path \( \vec{q}_2 = q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow \ldots \) in \( S_{TA} \) if and only if there is a corresponding path \( \vec{q}_{RS} = q_1 \rightarrow q_1 \rightarrow \ldots \rightarrow q \rightarrow q_{t_1-1} \rightarrow q' \rightarrow q_2 \rightarrow q_2 \rightarrow q_1 \rightarrow q_1 \rightarrow \ldots \rightarrow q_{t_2-1} \rightarrow q_2 \rightarrow q_3 \rightarrow \ldots \) in \( RS_{TA} \).
2. \( RS_{TA} \parallel RS_{TA'} \) is strongly bisimilar to \( RS_{TA} \parallel TA' \).

The first part gives the needed evidence that the use of \( RS_{TA} \) is actually enough to determine whether the language accepted by \( TA \) is empty. The second part indicates that the transitions corresponding to time progress behave just like ordinary synchronizations between the automaton, so from the semantical point of view those do not differ from ordinary transitions. The theorem also allows building the reduced state space of a system \( TA_1 \parallel \ldots \parallel TA_n \) in a compositional manner by first building the reduced state spaces of the individual components \( TA_i \) and afterwards combining those with parallel operator.

With the above results, we can now conclude the following:

**Corollary 4** Let \( TA_1, TA_2 \) and \( TA \) be weakly constrained timed automata.

1. The language accepted by \( TA \) is nonempty if and only if \( RS_{TA} \) contains a cycle which is reachable from the initial state, and contains at least one 1-transition, and a control state component \( s \in F \).
2. If \( TA \) is built from several parallel components, like \( TA_1 \) and \( TA_2 \), then the emptiness of \( TA \) can be checked by observing the existence of such a cycle from \( RS_{TA_1} \parallel RS_{TA_2} \).

The existence of a 1-transition in the cycle is required in order to ensure that the corresponding timed word is progressive.

These results show how the verification of dense time properties of a weakly constrained timed automaton can be reduced to cycle detection.
from a finite discrete structure which is suitable for algorithmic verification. However, some further optimizations can be achieved by noticing that all the state information except the fact whether a state is accepting or not, is actually superfluous for the verification task. So, the structure that is needed greatly resembles the labeled transition systems that are used as semantical models for process algebra descriptions. Actually, our structure, the reduced state space, can be seen as a labeled transition system where the states corresponding to acceptance states are augmented with atomic propositions \( f \).

Next we will rephrase a result concerning the truth of Linear temporal logics preserving reductions for labeled transition systems where states can be augmented with a set of atomic propositions (Kaivola, 1996). Let us call this structure Augmented labeled transition system.

In the theorem, the hide-operator well known in process algebra is also used. The semantics of hide is that it simply replaced all the listed transition labels with \( \tau \)-labels, denoting an internal action. The theorem also refers to the CFFD (Chaos Free Failures Divergences) semantical model (Valmari and Tienari, 1995), which is a semantics inducing an equivalence relation and the related pre-order between labeled transition systems. It is a variant of the Failures Divergences model well known from the CSP community.

**Theorem 5** Let \( \text{ALTS}_i = (S_i, A_i, R_i, s_0, L_i) \) be alts's for \( i \in \{1,2\} \), where \( L_i \) is a mapping from \( S_i \) to the set of atomic propositions true in that state.

1. If \( \text{ALTS}_1 \) and \( \text{ALTS}_2 \) are CFFD-equivalent, then for every formula \( \varphi \) of Linear Temporal Logic without 'nexttime'-operator (LTL' in short): \( \text{ALTS}_1 \vdash \varphi \) if and only if \( \text{ALTS}_2 \vdash \varphi \).
2. If \( \varphi \) is a formula of LTL' that does not refer to actions in the set \( A' \), then: \( \text{ALTS}_1 \vdash \varphi \) if and only if \( (\text{hide} A' \text{ in } \text{ALTS}_1) \vdash \varphi \).
3. Let \( A'_1 \) and \( A'_2 \) be such that \( A'_1 \cap A'_2 = \emptyset \) and \( A'_2 \cap A'_1 = \emptyset \), and furthermore let \( \varphi \) be such a LTL' formula that does not refer to actions in \( A'_1 \cup A'_2 \). Now \( (\text{hide} A'_1 \cup A'_2 \text{ in } (\text{ALTS}_1 \parallel \text{ALTS}_2)) \vdash \varphi \) if and only if \( (\text{hide} A'_1 \text{ in } \text{ALTS}_1) \parallel (\text{hide} A'_2 \text{ in } \text{ALTS}_2) \vdash \varphi \).
4. If \( \text{ALTS}_1 \subseteq_{\text{CFFD}} \text{ALTS}_2 \) then for a LTL' formula \( \varphi \) if \( \text{ALTS}_2 \vdash \varphi \) then also \( \text{ALTS}_1 \vdash \varphi \).

This result thus covers the general case with a finite set of state propositions, but we need only one of them, \( f \), which is true for the acceptance states only.

Since the nonexistence of an acceptance cycle can be expressed in LTL' with the negation of the formula \( \Box \Diamond f \land \Diamond \Box \Diamond \text{executed}(1) \), we can use the above result in order to reduce the size of the structure before the cycle detection. Since the property we are verifying refers only to
atomic proposition \( f \) and to the transition label 1, it is possible to hide all the other labels. Theorem 3 allows the compositional construction of reduced state space, and because of part 3 of Theorem 5, hiding can be 'pushed' inside the components for the transitions labels that are internal to one component. Then before making the composition, thanks to part 1 of Theorem 5, these components may be minimized with respect to an equivalence relation that is stronger or equal to the CFFD, for example one suitable is the version of weak bisimulation equivalence that takes into account the divergence information. The next corollary summarizes this, as well as the fact that a CFFD-abstraction of a reduced state space can be used to guarantee the nonexistence of an acceptance cycle.

**Corrollary 6** Let \( TA \) and \( TA_{\overline{\varphi}} \) be weakly constrained timed automata, where the latter corresponds to negation of the property \( \varphi \), and let \( RA_{TA} \) and \( RA_{TA_{\overline{\varphi}}} \) be the reduced state spaces of the above. Furthermore let \( RS_{\min} \) be an augmented labeled transition system which is CFFD-equivalent with \( hide A_{TA} \cup A_{TA_{\overline{\varphi}}} \) in \( RS_{TA} \parallel RS_{TA_{\overline{\varphi}}} \). Now the following are equivalent:

1. \( TA \) satisfies the property \( \varphi \),
2. \( RS_{TA} \parallel RS_{TA_{\overline{\varphi}}} \) does not have any accepting cycles containing 1 transitions,
3. \( RS_{\min} \) does not have any accepting cycles containing 1 transitions.

Furthermore, let \( RS' \) be an augmented labeled transition system, for which the relation \( RS_{\min} \subseteq_{CFFD} RS' \) holds. If \( RS' \) does not have any accepting cycles containing 1 transitions then \( TA \) satisfies the property \( \varphi \).

The last part of the corollary is an interesting consequence of part 4 of Theorem 5. It allows the possibility of replacing some components in the reduced state space with ones that are 'bigger' with respect to CFFD pre-order, and if the reduced state space with this 'bigger' version (often called *abstraction*) does not contain cycles, it is guaranteed that the concrete system does not contain a cycle either.

As the following theorem shows, the nonexistence of an acceptance cycle can be verified when the time progress transitions are hidden.

**Theorem 7** Let \( TA \) and \( TA_{\overline{\varphi}} \) be weakly constrained timed automata, and \( RS' = hide A_{TA} \cup A_{TA_{\overline{\varphi}}} \cup \{1\} \) in \( RS_{TA} \parallel RS_{TA_{\overline{\varphi}}} \). If \( RS' \) does not contain any accepting cycles then \( TA \) satisfies the property \( \varphi \).

Thus, the cycle detection might at first be done for a reduced state space that has no visible transitions at all, and only if a cycle is found, then the time progress transition 1 is unhidden and the existence of the cycle is re-checked.

If only some of the system components actually refer to the progress of time, some further optimization can be achieved. This is formalized with the following theorem.
Theorem 8 Let $T A_1$ be an arbitrary timed automaton, $T A_2$ a timed automaton that allows time to progress in every state, and $R S_{T A_1}, R S_{T A_2}$ the corresponding reduced state spaces. Let us denote with $R S'_{T A_2}$ the alt which is obtained from $R S_{T A_2}$ by removing all the 1 transitions. Now $R S_{T A_1} R S_{T A_2} = C F D R S'_{T A_1} R S_{T A_2}$.

4. FISCHER’S PROTOCOL

Fischer’s mutual exclusion protocol uses only one shared variable in guaranteeing the mutual exclusion, but it requires correct timing from the processes trying to enter their critical sections.

Every client of the protocol is given a unique identifier $id$ which is a natural number greater than zero. When a client wants to enter the critical section, it first waits until the value of the shared variable is zero which indicates that nobody is currently in the critical section. When detecting that, the client writes its own $id$ to the shared variable. This operation has to be done within $T$ time units from the moment when the value of the shared variable was noted as zero. After that, the client waits for $T'$ time units, where $T' > T$. If the shared variable still contains the $id$ of the client, it is allowed to enter the critical section. In other cases, the mutex-protocol is started from the beginning.

Every process as well as the memory of the system are modeled as timed automata. The automata $P_i$ and $M e m$, modeling process $i$ and the shared memory are defined in Figure 1. As usual, we assume that all the states in the automata describing the system are accepting.

\[ P_1; ...; P_n || M e m \]

\[ T \leq x_1 \rightarrow \text{read} \]

\[ x_1 = 0 \rightarrow \text{write} \]

\[ x_1 > 0 \rightarrow \text{read} \]

\[ x_1 \leq 0 \rightarrow \text{write} \]

\[ T' \leq y \rightarrow \text{read} \]

\[ y = 0 \rightarrow \text{write} \]

\[ y > 0 \rightarrow \text{read} \]

\[ y \leq 0 \rightarrow \text{write} \]

Figure 1. Fischer’s protocol

A timed automaton describing the behavior of the system is obtained with the parallel composition $P_1 || ... || P_n || M e m$, where $||$ denotes the full interleaving operator.

The negation of the mutual exclusion property, which states that there is maximally one process at a time in the critical section, is captured by the automaton $M u t e x$ which is defined in Figure 2.

The protocol satisfies the mutual exclusion property if we can show that the intersection of the languages defined by $P_1 || ... || P_n || M e m$ and $M u t e x$ is empty.
In order to do the verification we produced the transition systems $RS_{Mutex}$, $RS_{Mem}$ and $RS_{P_i}$ for every $i \in \{1, \ldots, n\}$. The results with parameter values $T = 1$, $T' = 2$ and $n = 2$ are shown in Figure 3.

Then the next step was building the transition systems

$$
System := red(\text{red}(\text{hide} \text{ read, write, req, 1 in red}(RS_{P_1}) \parallel \cdots \parallel \text{red}(RS_{P_n})) \parallel \text{red}(\text{hide 1 in } RS_{Mem}))
$$

$$
Test := \text{red}(\text{hide enter, leave in System } \parallel \text{red}(\text{hide 1 in } RS_{Mutex}))
$$

where $\text{red}$ denotes a reduction procedure that preserves CFFD-equivalence. Transition systems $System$ and $Test$ with the example parameters are shown in Figure 4.

Now because of Corollary 6 and Theorem 8 it follows that if $Test$ does not have any accepting cycles then the intersection of the languages defined by $(P_1 \parallel \cdots \parallel P_n) \parallel Mem$ and $Mutex$ is empty, and thus the protocol satisfies the mutual exclusion property. Since all the automata
are weakly constrained, the verification results hold for the dense time domain.

We verified the system with several parameter value combinations. The ARA (Advanced Reachability Analysis) toolset (Valmari et al., 1993) was used for the verification. The sizes of the largest transition systems that were encountered during the verification are shown in Table 1. For the compositional style of building the state space it is often the case that not the final result, but some intermediate step is the one that produces the largest transition system, thus the one that restricts the size of the systems that the tool can handle. So, for the compositional method, the largest intermediate result is shown in the table. Besides the number of client processes, the time parameter values have a great impaction the size of the verification model. This fact is also reflected in Table 1.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T$</th>
<th>$T'$</th>
<th>compositional</th>
<th>noncompositional</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>294</td>
<td>415</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>496</td>
<td>777</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1882</td>
<td>3351</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1417</td>
<td>1899</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3013</td>
<td>4559</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>19340</td>
<td>33399</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>6901</td>
<td>7205</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>18838</td>
<td>27769</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>200011</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>34124</td>
<td>43854</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>120254</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>172115</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>156564</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. Sizes of the reduced state spaces with the various parameter combinations. Note that the 8 client case was done with an optimized version of the protocol clients, where the enter, req and leave transitions were not used, but instead just the information whether the client is in its critical section or not.
Let us now demonstrate what happens if the property to be verified does not hold. Consider the protocol in case of two clients and parameter values $T = 1$, and $T' = 2$ for $P_1$ but $T = T' = 1$ for $P_2$. The transition systems $System$ and $Test$ are shown in Figure 5.

![System and Test in the case of an erroneous parameter value in the second entity](image)

*Figure 5. System and Test in the case of an erroneous parameter value in the second entity*

As is seen, the $Test$ now contains an accepting cycle, so the property might not hold. The next step is to build $System'$ and $Test'$ which are just as the above two, except the transition label 1 is not hidden. The result $Test'$, which is shown in Figure 6, indeed contains an accepting cycle, with a 1-transition, so Corollary 6 now says that this version of the protocol does not guarantee the mutual exclusion.

![Test' showing the existence of an accepting cycle with progress of time](image)

*Figure 6. Test' showing the existence of an accepting cycle with progress of time*

It is impossible to see the reason for the error from $Test'$, and the only way to produce a useful counter-example is trying to preserve more transition information in the transition system from which the cycle is to be detected.

In comparison with other reported studies on Fischer’s protocol, our compositional method performs quite well. The Discrete Time SPIN (Bośnacki and Damm, 1998) was able to do the verification up to 6 client processes, and the Real Time Spin (Tripakis and Courcoubetis, 1996) with 4 clients and UPPAAL with 8 clients. The tool Kronos has been reported to manage with 5 clients (Daws et al., 1996), and UPPAAL with 8 (Larsen et al., 1995). With the convex-hull over approximation tech-
nique (Daws and Tripakis, 1998) both UPPAAL and Kronos can handle the protocol with 9 clients. In the above case studies, the timing parameters $T$ and $T'$ used were not reported, but since those methods, except Discrete Time SPIN, rely on region or zone techniques, it is presumable that the results are not as sensitive to the values of time parameters as ours.

5. **BOUNDED RETRANSMISSION PROTOCOL**

The bounded retransmission protocol, or BRP in short (D’Argenio et al., 1997), is used to implement a reliable file transfer service using an unreliable point-to-point data link, such as a telephone line. The underlying data link offers unreliable delivery of small, fixed size data packets. The size of the packets is far smaller than a typical file which has to be transferred. Thus, the file has to be fragmented into frames, which are smaller than a data packet of the link.

The formal specification of the protocol consists of four timed automata: $\text{Sender}$, $\text{Receiver}$, $\text{Ch}_{SR}$, $\text{Ch}_{RS}$ where the latter two describe the behavior of the underlying channel. The description of the entire system is parallel composition of the automata.

Due to the restricted length of the paper, only the verification result is reported here, and we suggest that reader see the full version of the paper (M. Luukkainen, 2001) to get a more detailed view of the protocol and how the verification was done.

The parameter describing the length of the file in the $\text{Sender}$ automaton turned out to be problematic from the verification point of view. In fact, there has to be an upper limit in the length of the transmitted file in order to make $\text{Sender}$ finite stated. And even if a limit is set, it has a great impact on the size of the generated transition systems. Therefore, we used an abstraction $\text{Sender}'$ of the sender automaton. In $\text{Sender}'$ the actual length of the transferred file is not remembered, but instead it is ’guessed’ while sending the file.

Let us denote the sender with $k$ as the maximum length of files as $\text{Sender}_k$. Since for all values of $k$ the reduced state space $RS_{\text{Sender}_k}$ of the actual sender automaton is CFFD refinement of the transition system $RS_{\text{Sender}'}$ of the abstraction, i.e. $RS_{\text{Sender}_k} \subseteq_{\text{CFFD}} RS_{\text{Sender}'}$, the Corollary 6 guarantees, that if there are no acceptance cycles in the system with the abstract sender, the property holds for the actual version of the system. So, with the use of an abstraction of the sender, we now gain two things. Firstly there is a substantial saving in the size of the generated transition systems, since already the size of $\text{Sender}_3$ is
greater than that of the abstraction. Even if the effect of increasing $k$ appears to be linear, see the Table 2, it is advantageous to eliminate this source of state explosion. The second, even more impressive advantage is that the verification generalizes to all the possible file lengths.

$$
\begin{array}{|c|c|c|c|c|c|c|}
\hline
 & 2 & 4 & 6 & 8 & 10 & \text{Abstraction} \\
\hline
\text{Sender} & 51 & 107 & 163 & 219 & 275 & 65 \\
\text{Model} & 1156 & 2393 & 3628 & 4864 & 6100 & 1433 \\
\hline
\end{array}
$$

*Table 2.* Effect of the maximum file length to the size of sender and to the verification model in case the channel delay is 1 and number of retransmissions is 1

The correctness requirements of the protocol are twofold. The automaton defining the correspondence of indications given to clients of the protocol was quite straightforward to construct. The verification itself was then done in the same way as in the case of Fischer’s protocol.

From the verification point of view, the other requirement, *the frames are received in correct order, and no frames are dropped if an error is not indicated* was more problematic. To verify this property, the theory of data independence (Wolper, 1986) was exploited. The data independence guarantees that, if a transfer protocol does not alter or use in its logic the user data, it is enough to consider just a small number of different types of user data frames, even if the domain of the frames is extremely large. So, with the help of this we formulated the corresponding property automaton and carried out the verification in the same way as was reported for Fischer’s protocol.

The verification was done with several choices of values for the parameters $D_{\min}, D_{\max}, \text{and } MAX$ where the first two define the transmission delay of a frame, and the last describes the maximum number of retransmission trials for a frame. Table 3 summarizes the sizes of the largest reduced state spaces during the verification.

$$
\begin{array}{|c|c|c|c|c|c|}
\hline
D_{\min} & D_{\max} & MAX & \text{verifying indications} & \text{verifying data transfer} \\
\hline
0 & 1 & 1 & 1433 & 1993 \\
0 & 1 & 3 & 5415 & 7411 \\
1 & 2 & 1 & 2108 & 4499 \\
1 & 2 & 3 & 8915 & 11693 \\
2 & 3 & 1 & 2958 & 3748 \\
2 & 3 & 3 & 12967 & 16378 \\
\hline
\end{array}
$$

*Table 3.* Sizes of the largest encountered reduced state spaces with the various parameter combinations.
In D’Argenio et al., 1997 verification of the protocol with UPPAAL was reported. The main difference compared to our results is that in the study of D’Argenio et al., 1997 the verification was done only for files of length 1, 2 and 3 frames, and the correctness of data transfer was not verified. In the UPPAAL study the channel delay was fixed to one, and as in our case, the maximum number of retransmissions between 1 and 3 were considered.

In order to compare the efficiency of our method versus UPPAAL, we ran our verification with a Sender process where the maximum length of transmitted files was fixed to 3. With the number of retransmissions 1, 2 and 3 the running times with our method were 2, 4 and 13 seconds and with UPPAAL the corresponding measures were 61, 156 and 340 seconds. Our method also outperforms the Discrete Time SPIN verification reported in Bošnački and Damms, 1998.

6. CONCLUSIONS

In the article we showed how the untimed process algebra theory and tools can be used in verifying a category of dense time properties from a restricted class of timed automata. The work was based on the known result that by restricting ourselves to a class of timed automata where only the operators ≤, =, and ≥ are used in the clock comparisons, most of the interesting dense time properties are verifiable using discrete time methods. Those results were refined in order to fit the process algebraic context, and furthermore, some reduction strategies based on behavior equivalences, abstraction and compositionality were indicated. It is in particular the possibility for the compositional generation of state spaces that makes the process algebraic method efficient for the timed systems, as well.

The experimental results obtained from the verification of Fischer’s mutual exclusion protocol and Bounded retransmission protocol indicated that the efficiency of our method is of the same magnitude as for the other available timed methods. Especially in the BRP case study our method clearly worked faster than UPPAAL in the study reported in D’Argenio et al., 1997. An interesting point in our BRP study was the possibility of using an abstract version of the sender process that allowed us to conduct the verification for arbitrary long files. Thus, with our method, we were able to obtain a more complete verification result than the UPPAAL, since there the maximum length of files had a fixed upper bound of just 3 frames.
REFERENCES


TESTING LIVENESS PROPERTIES

Approximating Liveness Properties by Safety Properties

Ulrich Ultes-Nitsche and Simon St James
University of Southampton
Department of Electronics and Computer Science
Southampton, SO17 1BJ, United Kingdom
{uun,spstj99r } @ecs.soton.ac.uk

Abstract

Satisfaction within fairness establishes an abstract notion of truth under fairness in linear-time verification. It can be argued that the idea of an implementation meeting the intersection of the system behaviour and the requirement specification forms a desirable basis for testing systems. Unfortunately this intersection of the behaviours of the system and the requirements specification is not accessible directly and cannot be computed practically. It is, however, possible to derive some information of this intersection from a high-level (abstract) specification of the system. We will present in this paper an abstraction-based approach that will enable us to derive some test cases, even for liveness properties.

Keywords: Test case generation, satisfaction within fairness, liveness properties, behaviour abstraction.

1. INTRODUCTION

In [12], the concept of satisfaction within fairness is presented and adopted to testing systems. The basic idea is to test a system not against requirement specifications (properties) on their own but against the intersection of the requirements specification with the behaviour of the system. A similar observation was made by Petrenko [10] at the same conference (the observation was made in his presentation). Such an approach is more suited towards verification than towards testing. The reason being the accessibility of the above mentioned intersection. We simply have no direct access to an exhaustive system behaviour description of the implemented system. However, we normally will have a high-level system specification on which the implementation is based upon.

We will give in this paper a definition of a safety property that approximates a liveness property. Using this safety approximation will enable us to test for violation of liveness properties in principle. The established definition will be
similar to the notion of satisfaction within fairness in verification [6; 7; 12]. In order to overcome practical limitation of the introduced concepts, we will adopt an abstraction result for satisfaction within fairness that will make the results of this paper more applicable.

2. LINEAR-TIME VERIFICATION

We review linear satisfaction of properties very briefly in this section. Let \( A \) be the finite set of actions that a system can perform. By \( A_\omega \) we denote the set of all infinitely long sequences of actions in \( A \). A computation \( c \) of the system is an element of \( A_\omega \), i.e. \( c = a_1a_2a_3\ldots \) such that all \( a_i \) are in \( A \). The behaviour \( B \) of the system is then a subset of \( A_\omega \).\(^1\) A property \( P \), from an abstract point of view, is itself a subset of \( A_\omega \), representing all correct behaviours with respect to the intuitive meaning of the property [1]. Hence the system, represented by its behaviour \( B \), satisfies property \( P \) (written: \( B \models P \)) if and only if all computations in \( B \) are also in \( P \). This linear-time satisfaction of properties is defined by

\[
B \models P \iff B \subseteq P.
\]

3. EXHAUSTIVE TESTING

In contrast to the satisfaction relation for linear-time temporal verification, when testing a system with respect to a property \( P \) (\( P \), in this context, can be called a requirement specification), we consider finite behaviours: We can only observe a system for a finite amount of time, during which we only see a prefix of its (potentially) infinite behaviour. Let us define formally the relation that describes testing a system exhaustively:

Let \( A^* \) denote the set of all finitely long sequences of actions in \( A \). For a subset \( S \) of \( A_\omega \) let \( \text{pre}(S) \) denote the set \( \{ w \in A^* \mid \exists x \in A_\omega : wx \in S \} \) of finite prefixes of behaviours in \( S \). For an element \( s \) of \( A_\omega \), let \( \text{pre}(s) = \text{pre}(\{s\}) \). \( B \) satisfies \( P \) with respect to exhaustive testing, written \( B \vdash P \) in this paper, if and only if

\[
\text{pre}(B) \subseteq \text{pre}(P).
\]

Typically, properties are classified into two groups, which reflect either absence of errors or satisfactory progress. These two property classes are called safety and liveness properties respectively [1]:

\( P \) is a liveness property if and only if

\[
\forall w \in A^* : \exists x \in A_\omega : wx \models P.
\]

---

\(^1\) In terms of formal language theory, \( c \) is an \( \omega \)-word and \( B \) is an \( \omega \)-language on the alphabet \( A \).
$P$ is a safety property if and only if

$$\forall x \in A^\omega : (x \not\in P \Rightarrow \exists w \in \text{pre}(x) : \forall z \in A^\omega : wz \not\in P).$$

Note that any property is the intersection of a safety and a liveness property [1]. Also note that $P$ is a liveness property if and only if $\text{pre}(P) = A^*$ (recall that $P$ is a subset of $A^{\omega})$ [2]. Therefore $\text{pre}(B)$ is always a subset of $\text{pre}(P)$ for a liveness property $P$. In other words: We cannot test liveness properties, because they are always trivially satisfied (wrt. exhaustive testing).

In the case of safety properties, however, exhaustive testing is as precise as linear-time verification:

**Lemma 1** $P$ is a safety property if and only if $B \vdash P$ $\iff$ $B \models P$, for all $B$.

**Proof** “if:” We assume $B \vdash P \Rightarrow B \models P$ and take the counterpositive: $B \not\vdash P \Rightarrow B \not\vdash P$, for all $B$. Let $x \in A^\omega$ such that $x \not\in P$. Then, by the above assumption, $x \not\in P$. Let $w \in \text{pre}(x)$ such that $w \not\in \text{pre}(P)$. Then for all $z \in A^\omega$, $wz \not\in P$. As $x$ was chosen arbitrarily, $P$ is a safety property.

“only if:” Let $P$ be a safety property. Since $B \models P$ always implies $B \vdash P$, we only have to show $B \vdash P \Rightarrow B \models P$. We prove the counterpositive version: $B \not\vdash P \Rightarrow B \not\models P$. Assume $B \not\vdash P$. Then there must be an $x \in B$ such that $x \not\in P$. Because $P$ is a safety property, there must be a prefix $w$ of $x$ such that for all $z \in A^\omega$, $wz \not\in P$. Therefore $w \in \text{pre}(P)$. Because $w \in \text{pre}(B)$, we have $B \not\models P$, completing the proof. $\square$

4. **APPROXIMATING LIVENESS PROPERTIES BY SAFETY PROPERTIES**

We are going to define to a behaviour $B$ and a property $P$ a safety property $S_{BP}$. Testing $S_{BP}$ exhaustively will establish an approximation to the linear-time satisfaction of $P$ by $B$. Note that $P$ can be any property, e.g. a liveness property.

First, we need a dual operation to constructing prefix sets: the Eilenberg-limit. Let $L$ be a subset of $A^*$. The Eilenberg-limit $\text{lim}(L)$ [3; 11] is defined by

$$\text{lim}(L) = \{x \in A^\omega | |\text{pre}(x) \cap L| = \infty\}.$$

We define the safety property to $B$ and $P$ by

$$S_{BP} = \text{lim}(\text{pre}(B \cap P)).$$

First we prove that $S_{BP}$ is indeed a safety property:

**Lemma 2** $S_{BP}$ is a safety property.

**Proof** Let $x \in A^{\omega}$. If $x \not\models S_{BP}$, then there must be a prefix $w \in \text{pre}(x)$ that is not in $\text{pre}(B \cap P)$ (otherwise $x \models S_{BP}$). Then, for all $z \in A^{\omega}$,
wz \not\in \lim(\text{pre}(B \cap P)) \text{ and thus } wz \not\in S_{BP}. \text{ Hence } S_{BP} \text{ is a safety property.} \hfill \square

Note that \( \text{pre}(\lim(\text{pre}(B \cap P))) = \text{pre}(B \cap P) \). \text{ Hence } 

\[ B \vdash S_{BP} \iff \text{pre}(B) \subseteq \text{pre}(B \cap P). \]

This relation also occurs in the context of linear-time properties with an abstraction notion of fairness [7; 12]. We use the following result to interpret the definition of \( S_{BP} \) with respect to exhaustive testing:

**Lemma 3** Let \( B \subseteq A^{\omega} \) and \( P \subseteq A^{\omega} \) be the behaviour of a system and the property it should meet respectively. Then \( B \vdash S_{BP} \) if and only if \( B \cap P \) is a dense set in \( B \) [7; 12].

We use in this lemma the following definition of a metric space (known as the Cantor topology) (for topological notions see [4]):

Let \( \text{common}(a, b) \) designate the longest common prefix of two computations \( a \) and \( b \) in \( A^{\omega} \). We define the metric \( d(a, b) \) by

\[
\forall a, b \in A^{\omega}, a \neq b : d(a, b) = \frac{1}{|\text{common}(a, b)| + 1} \\
\forall b \in A^{\omega} : d(b, b) = 0.
\]

How can we interpret the obtained results? Lemma 3 tells us that testing safety property \( S_{BP} \) exhaustively is equivalent to checking whether \( B \cap P \) is a dense set in \( B \). However, in topological terms, \( B \cap P \) being a dense set in \( B \) is an approximation of \( B \subseteq B \cap P \) and therefore an approximation of linear time satisfaction \( B \vdash P \): A system whose behaviour satisfies that \( B \cap P \) is a dense set in \( B \), but not that \( B \vdash P \), must be observed for an infinite amount of time in order to detect \( B \not\vdash P \) [12]. This distinction is therefore practically irrelevant, since systems cannot be observed for an infinite amount of time. Because \( S_{BP} \) is a safety property, we can indeed test it. As \( P \) can even be a liveness property, we have found a way to check the (approximate) satisfaction of a (liveness) property by testing a related safety property exhaustively.

Lemma 3, establishing a very interesting theoretical result about the testability of liveness properties, is unfortunately not too practical without additional effort: As we need access to \( B \cap P \) in the definition of \( S_{BP} \), we must be able to construct the intersection of the implementation of a system \( (B) \) and the property \( (P) \). But the implementation will in practice not be accessible in such a way that this intersection could be constructed.

Having mentioned above that the relation \( \text{pre}(B) \subseteq \text{pre}(B \cap P) \) is a satisfaction relation in verification under fairness constraint (called satisfaction within fairness), we can use an abstraction result for this relation that will allow us to use the results of this section at least partly (in general we will lose some information, but not all).
5. SATISFACTION WITHIN FAIRNESS AND BEHAVIOUR ABSTRACTION

In this section we use behavior abstraction [6; 7]. Before we define it this, we need a few preliminary results and definitions. First of all we will write $B \models P$ for saying that $B$ satisfies $P$ within fairness:

$$B \models P \iff \text{pre}(B) \subseteq \text{pre}(B \cap P).$$

Let $A^\infty = A^* \cup A^\omega$. The leftquotient [3] of $M \subseteq A^\infty$ by $w \in A^*$ is then the set

$$\text{cont}(w, M) = \{v \in A^\infty \mid vw \in M\}.$$

**Lemma 4** $\lim(L) \models P$ if and only if

$$\forall w \in \text{pre}(\lim(L)) : \exists x \in \text{cont}(w, \lim(L)) : wx \in P [7].$$

We can now give a definition of behaviour abstraction and explore its relation to satisfaction within fairness. It is defined by alphabetic language homomorphisms extended to $\omega$-languages [5]:

**Definition 5** $h : A^\infty \to A^{\infty}$ is an abstraction homomorphism if and only if the following conditions hold:

- $h(A) \subseteq A' \cup \{\epsilon\}$  
  (action renaming and hiding)
- $\forall v, w \in A^* : x \in A^\omega : h(vw) = h(v)h(w)$ and $h(vx) = h(v)h(x)$  
  (compatibility with concatenation)
- $h(A^\omega) \subseteq A^\omega$  
  (undefined for $\omega$-words that would be taken to finite words).

Note that abstraction homomorphisms are partial mappings since they are not defined on $\omega$-words that would be taken to finitely long words. Also, note that they do not increase the length of words; that is, we always have that $|h(w)| \leq |w|$. For an abstraction homomorphism $h$, we call $\lim(h(L))$ the abstraction of concrete behaviour $\lim(L)$. For prefix-closed regular $L$, $\lim(h(L)) = h(\lim(L)) [7]$. We will consider subsequently only behaviours that are the Eilenberg-limit of a prefix-closed regular language $L$ (i.e. $B = \lim(L)$, $L = \text{pre}(L)$, and $L$ is regular). That means we are considering behaviours that can be represented by finite-state labelled transition systems.

In general, a property satisfied within fairness on the abstraction is not necessarily satisfied within fairness on the concrete behaviour in a corresponding way. We need an additional requirement to be satisfied by the abstraction homomorphism [8; 9]:

$$\forall w \in \text{pre}(\lim(L)) : \exists x \in \text{cont}(w, \lim(L)) : wx \in P [7].$$
Essential information about properties that are satisfied within fairness by \( \lim(L) \) is contained in the sets \( \text{cont}(w, L) \), for \( w \in L \). At the abstract level, we obviously have access to \( \text{cont}(h(w), h(L)) \), but we really need \( h(\text{cont}(w, L)) \) in order to ensure that properties satisfied within fairness by the abstraction will also be satisfied within fairness by the concrete system in a corresponding way. Thus, we need to find conditions under which \( \text{cont}(h(w), h(L)) \) can be used instead of \( h(\text{cont}(w, L)) \).

In general, \( h(\text{cont}(w, L)) \) is a proper subset of the set \( \text{cont}(h(w), h(L)) \). In order to obtain sufficient information about \( h(\text{cont}(w, L)) \) from considering \( \text{cont}(h(w), h(L)) \), one would be tempted to require equality of the two sets. Those homomorphisms are continuation closed, since computation of the continuation (leftquotient) and the abstraction commute. However, this is stronger than needed. Indeed, since we are dealing with satisfaction within fairness, it is sufficient that the computations in \( \text{cont}(h(w), h(L)) \) “eventually” become computations in \( h(\text{cont}(w, L)) \):

**Definition 6** \( h \) is weakly continuation-closed on \( L \) if and only if for all \( w \in A^* \), there exists \( v \in \text{cont}(h(w), h(L)) \) such that \( \text{cont}(v, \text{cont}(h(w), h(L))) = \text{cont}(v, h(\text{cont}(w, L))) \).

One should not worry about this rather technical definition. Its essence of this definition revealed by the following lemma:

**Lemma 7** Let \( L \) be regular. If \( h(L) \) does not contain maximal words,\(^2\) then the condition

\[
\lim(h(L)) \models P \text{ if and only if } \lim(L) \models h^{-1}(P)
\]

holds if and only if \( h \) is weakly continuation-closed on \( L [6] \).

Weakly continuation-closed homomorphisms are therefore precisely the class of homomorphisms that preserve satisfaction within fairness.

What is the essence of this result? To test whether the concrete behaviour \( \lim(L) \) (i.e. the implementation’s behaviour) satisfies property \( h^{-1}(P) \)\(^3\) with the help of a safety approximation \( S_{\lim(L)h^{-1}(P)} \) — a test that is practically impossible directly — we can test instead whether the abstract behaviour \( \lim(h(L)) \) satisfies abstract property \( P \) with the help of \( S_{\lim(h(L))P} \) which can be constructed effectively. Since we can turn an observed (finite) behaviour of the system under test on-the-fly into its corresponding abstract behaviour by application of abstraction homomorphism \( h \), we can then immediately test whether

---

\(^2\) \( w \in L \) is a maximal word in \( L \) if and only if \( \text{cont}(w, L) = \{ \varepsilon \} \). This restriction can be removed easily by introducing additional “dummy” actions [6]. However, we avoid introducing additional notation here.

\(^3\) Note that \( h^{-1}(P) \) can even be a liveness property.
the obtained trace is in $\text{pre}(S_{\text{lim}}(h(L)))^P$. If we tested the on-the-fly abstraction of the system traces exhaustively against $S_{\text{lim}}(h(L))^P$ would immediately establish $\text{lim } (L) \models h^{-1}(P)$. We therefore have established an indirect way, with the help of abstraction, for testing even liveness properties of a system.

6. CONCLUSIONS

We have presented how liveness properties can be approximated by safety properties. The prefixes of these safety properties form then test cases that allow to check whether a system nearly meets some liveness properties. “Nearly” means in this context that if the liveness property were not satisfied (wrt. linear-time verification), then it would only be detectable by observing the system for an infinite amount of time and would therefore be practically irrelevant.

Since constructing the safety property mentioned above requires construction of the intersection of the system implementation and the original property, the described approach is not yet feasible. However an abstraction result as presented can be used that allows to gain at least some benefit from the defined safety approximation. The abstraction result in the previous section requires the concrete behaviour (i.e. the implementation) to be representable by a finite-state labelled transition system and the abstraction to be weakly continuation-closed. These are two conditions that will not necessarily be satisfied in practice, or we may simply not know whether they are satisfied when testing the system. However, if the system is an implementation derived from some high-level specification where the high-level specification has a finite-state labelled transition system semantics, we can simply assume that the high-level specification is a weakly-continuation closed abstraction of the implementation and that the implementation itself also has a finite-state labelled transition system semantics. If the assumption is false, we will have lost information and our testing will inaccurate, even though it may seem to be exhaustive. We will still be able to spot some problems with respect to meeting progress properties that will not be testable without the presented approach.

The claim made is obviously quite vague, even though it is underpinned by interesting theoretical results, and needs solid experimental justification. Experiments using the concepts introduced in this paper will be part of our future work. We will therefore aspire co-operation with researchers who work on the practical application of system testing. Any comments and suggestions are welcome.

REFERENCES


SVL: A SCRIPTING LANGUAGE FOR COMPOSITIONAL VERIFICATION

Hubert Garavel, Frédéric Lang*
INRIA Rhône-Alpes, 655, avenue de l’Europe, F-38330 Montbonnot St. Martin, France
Hubert.Garavel@inria.fr, Frederic.Lang@inria.fr

Abstract
Compositional verification is a way to avoid state explosion for the enumerative verification of complex concurrent systems. Process algebras such as LOTOS are suitable for compositional verification, because of their appropriate parallel composition operators and concurrency semantics. Extending prior work by Krimm and Mounier, this article presents the SVL language, which allows compositional verification of LOTOS descriptions to be performed simply and efficiently. A compiler for SVL has been implemented using an original compiler-generation technique based on the Enhanced LOTOS language. This compiler supports several formats and tools for handling Labeled Transition Systems. It is available as a component of the CAdP toolbox and has been applied on various case-studies profitably.

Keywords: Abstraction, Bisimulation, Compositional Verification, Concurrency, Coordination Language, E-LOTOS, Enumerative Verification, Labeled Transition System, LOTOS, Model-Checking, Process Algebra, Reachability Analysis, Specification, Validation.

1. INTRODUCTION

Enumerative verification (also called reachability analysis or model-checking) is a popular technique for verifying concurrent systems. Roughly speaking, it is a “brute force” technique, which consists in exploring and checking all states and transitions reachable by a concurrent system. This technique is confronted to the state explosion problem, which occurs when the number of states grows exponentially as the number of concurrent processes in the system increases. To avoid or reduce state explosion, various approaches have been proposed, among which:

*This work has been partially supported by DYADE, the BULL/INRIA joint venture for advanced research.
symbolic verification, on-the-fly verification, partial orders, symmetries, data flow analysis, and compositional verification.

This article deals with the latter approach (also known as *compositional reachability analysis* or *compositional minimization*). This approach assumes that the concurrent system under study can be expressed as a collection of communicating sequential processes, the behaviors of which are modeled as finite state machines or labeled transition systems (LTSs, for short). The sequential processes are composed in parallel, either in a flat or hierarchical manner.

In its simplest form [9, 23, 27, 32, 28, 29, 30], compositional verification consists in replacing each sequential process by an *abstraction*, simpler than the original process but still preserving the properties to be verified on the whole system. Quite often, abstracting a process is done by minimizing its corresponding LTS modulo an appropriate equivalence or preorder relation (e.g., a bisimulation relation, such as strong, branching or observational equivalence). If the system has a hierarchical structure, minimization can also be applied at every intermediate level in the hierarchy. Clearly, this approach is only possible if the parallel composition is “compatible” with LTS minimization: in particular, this is the case with the parallel composition operators of most process algebras, for which bisimulation is a congruence (see [32] for a discussion on this issue).

Although this simple form of compositional verification has been applied successfully to some complex systems (e.g., [11, 4] in the case of the LOTOS language [20]), it may be counter-productive in some other cases: generating the LTS of each process separately may lead to state explosion, whereas the generation of the whole system of concurrent processes might succeed if processes constrain each other when composed in parallel.

This issue has been addressed by refined compositional verification approaches [15, 6, 31, 7, 8, 18, 22, 5, 14], which allow to generate the LTS of each separate process by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioral restrictions imposed on each process by synchronization with its neighbor processes. Taking into account the environment of each process allows to eliminate states and transitions that are not reachable in the LTS of the whole system. Depending on the approach, interface constraints can be either written by the user or generated automatically.

The refined approach to compositional verification has been implemented in two tools, namely the TRACTA tool [14] and the PROJECTOR/DES2AUT tools [22]. The latter tools are part of the CADP protocol engineering toolbox [10] and have been applied to in-
dustrial case-studies [22, 24, 19]. Although positive, these experiments revealed various issues and shortcomings that prevented compositional verification to be used on a larger scale, especially in industrial projects. To solve these problems, we designed a scripting language named SvL, which can be seen as a process algebra extended with operations on LTSSs, e.g., minimization (also called reduction), abstraction, comparison, deadlock/livelock detection, etc. We implemented a compiler for this language, with the goal of making compositional verification easier for non-experts.

This article is organized as follows. Section 2 gives a few preliminary definitions. Section 3 briefly presents the principles and limitations of the DES2AUT tool of [22]. Section 4 defines the syntax and semantics of the SvL language. Section 5 introduces high-level features of SvL, which allow sophisticated strategies for compositional verification. Section 6 describes the implementation of the SvL 2.0 compiler. Finally, Section 7 gives concluding remarks and lists directions for future work.

2. DEFINITIONS

Labelled Transition Systems are the natural model for action-based specification languages, especially process algebras such as LOTOS [20]. Formally, an LTS is a tuple $M = (S, A, T, s_0)$, where $S$ is the set of states, $A$ the set of actions (or labels), $T \subseteq S \times A \times S$ the transition relation, and $s_0 \in S$ the initial state. A transition $(s, a, s') \in T$ indicates that the system can evolve from state $s$ to state $s'$ by performing action $a$. In enumerative verification, there are essentially two ways to represent an LTS:

- An explicit LTS is defined in extension, by enumerating all its states and transitions. Practically, there exist several formats to store explicit LTSS in computer files. The CADP verification tool set uses three such formats: AUT, a simple textual format, BCG (Binary Coded Graphs), a compact binary format based upon dedicated compression algorithms, and SEQ, a human-readable format for displaying sequences of transitions produced by verification tools to explain why a given property is not verified. There exist other LTS formats, for instance the Fc2 format used by the Fc2TOOLS [3], with which the CADP tools are interfaced.

- An implicit LTS is defined in comprehension by giving its initial state $s_0$ and its successor function $\text{succ} : S \rightarrow 2^T$ defined by $\text{succ}(s) = \{(s, a, s') \mid (s, a, s') \in T\}$. A generic representation of implicit LTSS is provided by the language-independent environment OPEN/CÆSAR [13] embedded in CADP. OPEN/CÆSAR offers primitives for accessing the initial state of an LTS and for enumer-
The tools available in CADP and Fc2Tools allow to perform the usual operations on (explicit or implicit) LTSs in several complementary ways, as summarized in the following table:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALDEBARAN</td>
<td>Reduction, comparison, deadlock/livelock detection</td>
</tr>
<tr>
<td>BC1G_IO</td>
<td>Conversion from one explicit LTS format to another</td>
</tr>
<tr>
<td>BC1G_LABELS</td>
<td>Hiding and renaming of labels</td>
</tr>
<tr>
<td>BC1G_M1N</td>
<td>Reduction</td>
</tr>
<tr>
<td>BC1G_OPEN</td>
<td>Implicit LTS view of an explicit LTS</td>
</tr>
<tr>
<td>CESAR.ADT, CESAR</td>
<td>LTS generation from a LOTOS description</td>
</tr>
<tr>
<td>CESAR.OPEN</td>
<td>Implicit LTS view of a LOTOS description</td>
</tr>
<tr>
<td>EVALUATOR</td>
<td>Deadlock/livelock detection</td>
</tr>
<tr>
<td>EXHIBITOR</td>
<td>Deadlock detection</td>
</tr>
<tr>
<td>EXP2Fc2</td>
<td>Conversion from EXP format to Fc2</td>
</tr>
<tr>
<td>EXP.OPEN</td>
<td>Implicit LTS view of an EXP description</td>
</tr>
<tr>
<td>FC2EXPLICIT, FC2IMPLICIT</td>
<td>Reduction, comparison, deadlock/livelock detection</td>
</tr>
<tr>
<td>GENERATOR</td>
<td>Explicit LTS generation from an implicit LTS</td>
</tr>
<tr>
<td>PROJECTOR</td>
<td>Abstraction (see Section 3 below)</td>
</tr>
</tbody>
</table>

3. THE DES2AUT TOOL

DES2AUT is a tool performing compositional generation of an LTS from a composition expression (called behavior) written in the DES language [22]. Given a behavior, the DES2AUT tool generates and minimizes its LTS compositionally modulo a relation $R$ (e.g., strong or branching bisimulation) specified on the command-line. The DES language is defined by the following grammar, where $B, B_0, B_1, B_2$ are non-terminal symbols denoting behaviors and $F, P, G_1, \ldots, G_n$ are terminal symbols denoting respectively file prefixes, LOTOS process identifiers, and LOTOS gate identifiers:

\[
B ::= F.aut \mid F.exp \mid F.lotos \mid \text{Proc}(F.lotos, P[G_1, \ldots, G_n]) \quad (1) \\
| \quad \text{hide } G_1, \ldots, G_n \text{ in } B_0 \quad (2) \\
| \quad B_1 \text{ if } [G_1, \ldots, G_n] \text{ else } B_2 \quad (3) \\
| \quad B_1 \text{ if } \neg[G_1, \ldots, G_n] \text{ then } B_2 \text{ else } B_1 \text{ if } \neg[G_1, \ldots, G_n] \quad (4)
\]

The syntactic constructs above denote:

(1) an explicit LTS, either contained in a file in AUT format, or generated from an implicit LTS given as an EXP description, an entire
LOTOS description, or a particular process $P$ defined in a LOTOS description (this process being instantiated with actual gate parameters $G_1, \ldots, G_n$);

(2) the behavior $B_0$ in which gates $G_1, \ldots, G_n$ are hidden using the LOTOS hiding operator;

(3) the behaviors $B_1$ and $B_2$ executing in parallel with synchronization and communication on gates $G_1, \ldots, G_n$ according to the LOTOS parallel composition semantics;

(4) the behavior $B_1$ restricted by synchronizing it on the set of gates $G_1, \ldots, G_n$ with $B_2$ ($B_2$ is considered as an automaton, the regular language of which expresses interface constraints); the resulting behavior is called the abstraction of $B_1$ w.r.t. $B_2$. The "?" symbol, if present, indicates that the user is unsure that the interface constraints expressed in $B_2$ are sound w.r.t. the real environment of $B_1$: as a consequence, validity checks are performed when the resulting LTS is composed in parallel with other LTSs.

Although the usefulness of the DES2AUT tool has been established on significant case-studies [22, 24, 19], its applicability is limited by several practical shortcomings:

- It only supports a single verification strategy: for each parallel composition operator, the operands are minimized first, then combined in parallel. Therefore, operands can only be combined two by two according to the evaluation order defined by the algebraic parallel composition operators; it is not possible to combine more than two components simultaneously. In some cases, this leads to state explosions that could be avoided otherwise. In the following example,

$$\text{(User1.aut | | | User2.aut) | [G] | Medium.aut}$$

generating the interleaved combination of "User1.aut" and "User2.aut" first—indeed from "Medium.aut" — is likely to produce a larger state space than if "Medium.aut" was taken into account.

So far, this problem has been circumvented by using other CADP tools (namely ALDÉBARAN and EXP.OPEN), which allow several LTSs to be combined together using LOTOS parallel composition operators. Practically, this is tedious because the DES and EXP languages accepted by DES2AUT and ALDÉBARAN-EXP.OPEN, respectively, are incompatible in syntax and semantics, and because many auxiliary DES and EXP files are needed to describe a compositional verification scenario this way. The problem is even worse
as there are often many possible scenarios corresponding to various decompositions of the system into sub-systems, which requires a trial-and-error approach to find the most effective decomposition.

- The DES2AUT tool relies on ALDÉBARAN to perform LTS minimization and does not support other bisimulation tools such as BCG_MIN and Fc2TOOLS. Moreover, it only accepts the AUT format, which is much less compact than the BCG format, so that compositional verification often aborts due to a lack of disk space.

- When a compositional verification scenario fails (e.g., if LTS generation, minimization, or parallel composition aborts at some point, because of a lack of memory or disk space, for instance), localizing and understanding the reason of the problem is difficult, as DES2AUT does not provide sufficient debugging information.

### 4. THE SVL LANGUAGE

To address these problems, we propose a scripting language for compositional verification. This language, named SVL, contains all the features of the Des language, as well as new ones that provide increased flexibility in the verification task.

To define the abstract syntax of SVL, we first introduce several terminal symbols: $F$ denotes a file prefix, $G, G_1, \ldots, G_n$ denote LOTOS gate identifiers, $L, L_1, \ldots, L_n$ denote (UNIX-like) regular expressions on labels considered as character strings, and $P$ denotes a LOTOS process identifier. We also introduce the following non-terminal symbols:

\[
\begin{align*}
\text{op} &::= \leq | \geq | = \\
\text{par} &::= \left[G_1, \ldots, G_n\right] | | | | \\
E &::= \text{aut} | \text{bcg} | \text{exp} | \text{fc2} | \text{lotos} | \text{seq} \\
R &::= \text{branching} | \text{strong} | \text{observational} | \text{safety} | \tau^*.a | \ldots \\
M &::= \text{std} | \text{fly} | \text{bdd} \\
T &::= \text{aldebaran} | \text{bcg.min} | \text{evaluator} | \text{exhibitor} | \text{fc2tools} | \ldots \\
U &::= \text{user} | <\text{empty}> \\
\end{align*}
\]

where op denotes an equivalence or preorder relation, par a LOTOS parallel operator, E a file extension, R a bisimulation relation, M an algorithmic method to compute bisimulations ("std" meaning a standard partition refinement algorithm such as Paige-Tarjan or Kanellakis-Smolka, "fly" meaning the Fernandez-Mounier on-the-fly algorithm, and "bdd" meaning an algorithm based on Binary Decision Diagrams), and T a tool.

The two main non-terminal symbols are $S$ (statements), and $B$ (behaviors). They are defined by the following grammar, where ‘[’ and ‘]’ delimit optional clauses in the grammar, and where ‘[’ and ‘]’ denote
the terminal bracket symbols. An SVL program is a list of statements “$S_1; \ldots; S_n$”.

$S ::= "F.E" = B_0$  \hspace{1cm} (S1)

| $"F.E" = R \text{ comparison } [\text{using } M] [\text{with } T] B_1 \text{ op } B_2$  \hspace{1cm} (S2)
| $"F.E" = \text{ deadlock } [\text{with } T] \text{ of } B_0$  \hspace{1cm} (S3)
| $"F.E" = \text{ livelock } [\text{with } T] \text{ of } B_0$  \hspace{1cm} (S4)

$B ::= "F.E" | "F.lotos": P[G_1,\ldots,G_n]$  \hspace{1cm} (B1)

| hide $L_1,\ldots,L_n$ in $B_0$  \hspace{1cm} (B2)
| $B_1 \text{ par } B_2$  \hspace{1cm} (B3)
| rename $L_1 \rightarrow L'_1,\ldots,L_n \rightarrow L'_n$ in $B_0$  \hspace{1cm} (B4)
| generation of $B_0$  \hspace{1cm} (B5)
| $R \text{ reduction } [\text{using } M] [\text{with } T] \text{ of } B_0$  \hspace{1cm} (B6)
| $U \text{ abstraction } B_1 \text{ sync } G_1,\ldots,G_n \text{ of } B_2$  \hspace{1cm} (B7)

An SVL behavior denotes either an explicit or an implicit LTS, contrary to the DES language, in which every implicit LTS is immediately converted to an explicit one. Formally, the semantics of a behavior $B$ is given by a denotation function $\llbracket B \rrbracket_{\sigma}$, the result of which is either an (explicit or implicit) LTS file, a LOTOS file, a LOTOS process instantiation, or an EXP composition expression (i.e., a set of explicit LTSs combined together with hiding and parallel composition operators). The subscript $\sigma$ denotes a set of file extensions, which denote all acceptable formats in which the result of $\llbracket B \rrbracket_{\sigma}$ should be produced. The value of $\sigma$ is determined by the context in which $B$ will be used, and namely by the tools that will be applied to $B$, given that certain tools require certain formats for their inputs and outputs (for instance, the Fc2TOOLS only handle LTSs in the FC2 format). Hence, format conversions may be required at some places but, for efficiency reasons, should be avoided as much as possible. $\sigma$ always contains at least an explicit LTS format; if it contains more than one, a preferred LTS format noted $\text{pref}(\sigma)$ is selected in the following preference order: $\text{bcg}$, then aut, then $\text{fc2}$, then $\text{seq}$. A dedicated type-checking is done on SVL programs, mainly to distinguish between explicit and implicit LTSs: certain constraints on $\sigma$ (listed below) must be satisfied, otherwise type-checking errors are reported. The semantics of behaviors is the following:

(B1) denotes a behavior contained in a file (or a LOTOS process instantiation). If it is an implicit LTS file (resp., a LOTOS file), then exp or $\text{fc2}$ (resp., $\text{lotos}$) must belong to $\sigma$ If $E \in \sigma$ then $\llbracket B \rrbracket_{\sigma}$ returns $B$ else it converts the LTS contained in $B$ to $\text{pref}(\sigma)$ using $\text{BCG\_IO}$ or $\text{EXP\_FC2}$. LOTOS process instantiations are handled similarly as LOTOS files.

(B2) denotes the label hiding of an implicit or explicit LTS using the ALDÉBARAN, BCG\_LABELS, and EXP\_OPEN tools. This generalizes the LOTOS hiding operator by allowing regular expressions on
labels. Contrary to the DES language, in which all implicit LTSS are converted into explicit ones before hiding, SVL semantics preserves implicit LTSS as long as possible (according to the definition of EXP composition expressions). If $\sigma$ contains $\text{exp}$ or $\text{fc2}$ and all labels $L_1, \ldots, L_n$ are LOTOS gates and $B_0$ is a parallel composition of behaviors, then $\llbracket B \rrbracket_\sigma$ returns the EXP composition expression $\text{hide } L_1, \ldots, L_n$ in $\llbracket B_0 \rrbracket_{\{\text{exp, aut, bcg, fc2, seq}\}}$, else $\llbracket B \rrbracket_\sigma$ returns the conversion to $\text{pref } (\sigma)$ of $\llbracket B_0 \rrbracket_{\{\text{bcg}\}}$ in which labels matching one of $L_1, \ldots, L_n$ are hidden.

(B3) denotes the parallel composition of $B_1$ and $B_2$. $\sigma$ must contain $\text{exp}$ or $\text{fc2}$. $\llbracket B \rrbracket_\sigma$ returns the EXP composition expression $\llbracket B_1 \rrbracket_{\sigma'} \text{par } \llbracket B_2 \rrbracket_{\sigma'}$ where $\sigma' = \{\text{exp, aut, bcg, fc2, seq}\}$.

(B4) denotes the label renaming of an explicit LTS using the BCGLABELS tool, which supports UNIX-like regular expression matching and substring replacement. $\llbracket B \rrbracket_\sigma$ returns the conversion to $\text{pref } (\sigma)$ of $\llbracket B_0 \rrbracket_{\{\text{bcg}\}}$ in which labels are renamed as specified by the rules $L_1 \rightarrow L'_1, \ldots, L_n \rightarrow L'_n$.

(B5) denotes the conversion from an implicit LTS to an explicit one, which is computed using OPEN/CAESAR compilers and GENERATOR. In SVL, such conversions must be requested explicitly using the “generation” operator, unlike the DES language in which such conversions are automatic and cannot be avoided. Let $\alpha$ be $\llbracket B_0 \rrbracket_{\sigma \cup \{\text{exp, fc2, lotos}\}}$. If $\alpha$ is already an explicit LTS, then $\llbracket B \rrbracket_\sigma$ returns $\alpha$ else it returns the conversion of $\alpha$ to an explicit LTS of format $\text{pref } (\sigma)$.

(B6) denotes the reduction of an LTS modulo an equivalence relation $R$, using an algorithmic method $M$ and a tool $T$ (ALDEBARAN, BCGMIN, or FC2TOOLS). For short, we abbreviate this operator to $\text{RMT-reduction}$. $\llbracket B \rrbracket_\sigma$ returns the conversion to format $\text{pref } (\sigma)$ of the RMT-reduction of $\llbracket B_0 \rrbracket_{\sigma^T}$, where $\sigma^T$ is the set of the accepted input formats for reduction with tool $T$.

(B7) denotes the abstraction w.r.t. interface constraints, which is computed using OPEN/CAESAR compilers and PROJECTOR. $\llbracket B \rrbracket_\sigma$ returns the conversion to format $\text{pref } (\sigma)$ of the abstraction of $\llbracket B_2 \rrbracket_{\{\text{bcg, exp, fc2, lotos}\}}$ w.r.t. the interface $\llbracket B_1 \rrbracket_{\{\text{aut}\}}$. The “user” keyword has the same meaning as the “?” symbol of the DES Language.

The statements have the following effects:

(S1) stores in file $F.E$ (where $E \neq \text{lotos}$) either $\llbracket B_0 \rrbracket_{\{E\}}$ if $E$ denotes an explicit LTS format or $\llbracket B_0 \rrbracket_{\{\text{aut, bcg, exp, fc2, seq}\}}$ if $E$ denotes an implicit LTS format.
(S2) compares two LTSSs modulo an equivalence or preorder relation $R$, using an algorithmic method $M$ and a tool $T$ (Aldébaran or Fc2Tools). Formally, it compares $[B_1]_{\sigma_T}$ and $[B_2]_{\sigma_T}$, where $\sigma_T$ denotes the set of possible input formats accepted by $T$. The result stored in file $F.E$ (where $E \notin \{exp, lotos\}$) is a (set of) distinguishing path(s) if the comparison returns false or an empty path otherwise.

(S3) detects deadlocks using a tool $T$ (Aldébaran, Evaluator, Exhibitor, or Fc2Tools) in $[B_0]_{\sigma_T}$, where $\sigma_T$ is the set of possible input formats accepted by $T$. The result stored in file $F.E$ (where $E \notin \{exp, lotos\}$) is a (set of) path(s) leading to (one or all) deadlock state(s), if any, or an empty path otherwise.

(S4) checks for livelocks in a way similar to statement (S3).

5. META-OPERATIONS

SVL has so-called meta-operations, which allow various compositional reduction strategies to be written concisely; this has proven to be useful in many case-studies. Meta-operations extend the syntax of SVL behaviors as follows:

$$B ::= \ldots | A \text{ R reduction using } M \text{ with } T \text{ of } B_0$$

where the attribute $A$ is defined by:

$$A ::= \text{ leaf } | \text{ root leaf } | \text{ node}$$

Informally, meta-operations have the effect of propagating $RMT$-reductions automatically at various places in the algebraic term $B_0$. Depending on the value of $A$, they have different semantics:

- "leaf" means that an $RMT$-reduction must be applied to all leaves "$F.E$" but also to all subterms of $B_0$ that generate an explicit LTS (i.e., to all abstraction, renaming, hiding, generation, and reduction operators). Leaf reduction is not propagated through generation and reduction operators. Moreover, to maximize the use of the Exp format of CADP and to avoid converting Exp descriptions into explicit LTSSs (unless requested by the user), $RMT$-reduction is not applied to hiding operators whose operands are parallel compositions of LTSSs.

- "root leaf" is similar to "leaf" except that $B_0$ itself is also $RMT$-reduced at the top-level.

- "node" is similar to "leaf" except that $RMT$-reductions are also applied to all parallel composition operators in the abstract tree of $B_0$. This emulates the reduction strategy of the DES2AUT tool and is mainly implemented for backward compatibility purpose.
Formally, “A R reduction using M with T of B” is expanded into \( \varepsilon_{RMT}(B, A) \) where \( \varepsilon_{RMT} \) is defined as follows and where the shorthand notation \( R_{RMT}(B) \) stands for an RMT-reduction of B:

\[
\varepsilon_{RMT}(B, A) = \text{if } A = \text{“root leaf” then } R_{RMT}(\varepsilon_{RMT}(B), \text{leaf}) \text{ else case } B \text{ in}
\]

- rename \( B \) case in \( B_0 \rightarrow R_{RMT}(\text{rename} \ldots \text{ in } \varepsilon_{RMT}(B_0, A)) \)
- if \( U \text{ abstraction } B_1 \ldots \text{ of } B_2 \rightarrow R_{RMT}(U \text{ abstraction } B_1 \ldots \text{ of } \varepsilon_{RMT}(B_2, A)) \)
- if \( A = \text{“node”} \) then \( R_{RMT}(\varepsilon_{RMT}(B_1, A) \text{ par } \varepsilon_{RMT}(B_2, A)) \)
- else \( \varepsilon_{RMT}(B_1, A) \text{ par } \varepsilon_{RMT}(B_2, A) \)
- hide \( L_1, \ldots, L_n \) in \( B_0 \rightarrow \text{if } \varepsilon_{RMT}(B_0, A) \text{ has the form } “B_1 \text{ par } B_2” \text{ and all } L_i \text{ are LOTOS gates then } hide L_1, \ldots, L_n \text{ in } \varepsilon_{RMT}(B_0, A) \)
- else \( R_{RMT}(\text{hide } L_1, \ldots, L_n \text{ in } \varepsilon_{RMT}(B_0, A)) \)
- otherwise \( \rightarrow R_{RMT}(B) \)

After meta-operation expansion, all behaviors containing meta-operations are replaced by simple behaviors (as defined in Section 4). Since the application of \( \varepsilon_{RMT} \) may create superfluous operations, an optimization function called \( \mathcal{O} \) is applied to the resulting behaviors in order to increase efficiency. For instance, \( \mathcal{O} \) merges nested hiding operators into a single hiding operator. \( \mathcal{O} \) also replaces \( R_{RMT}(R_{R'M'T'}(B)) \) by \( R_{R'M'T'}(B) \) if the relation \( R' \) is weaker or equal to the relation \( R \) (noted \( R' \sqsubseteq R \)), since an RMT-reduction is useless after an R'M'T'-reduction. At last, \( \mathcal{O} \) suppresses an RMT-reduction applied to the operand of a hiding (resp. abstraction) operator if the result of this hiding (resp. abstraction) itself is to be reduced modulo the same relation \( R \).

\[ \mathcal{O}(B) = \text{case } B \text{ in}
\]

- \( "F, E"[P[\ldots]] \rightarrow B \)
- \( B_1 \text{ par } B_2 \rightarrow \mathcal{O}(B_1) \text{ par } \mathcal{O}(B_2) \)
- generation of \( B_0 \rightarrow \text{generation of } \mathcal{O}(B_0) \)
- rename \( \ldots \text{ in } B_0 \rightarrow \text{rename } \ldots \text{ in } \mathcal{O}(B_0) \)
- if \( U \text{ abstraction } B_1 \ldots \text{ of } B_2 \rightarrow \mathcal{O}(B_1) \ldots \text{ of } \mathcal{O}(B_2) \)
- hide \( L_1, \ldots, L_n \) in \( B_0 \rightarrow \text{if } B_0 \text{ has the form } “\text{hide } L_{n+1}, \ldots, L_{n+m} \text{ in } B_0'” \text{ then}
\]
  \( \mathcal{O}(\text{hide } L_1, \ldots, L_{n+m} \text{ in } B_0') \)
  \( \text{else hide } L_1, \ldots, L_n \text{ in } \mathcal{O}(B_0) \)
- \( R_{RMT}(B_0) \rightarrow \text{case } B_0 \text{ in}
\]

- \( R_{R'M'T'}(B_0') \text{ where } R' \sqsubseteq R \rightarrow \mathcal{O}(R_{R'M'T'}(B_0')) \)
- \( \mathcal{O}(R_{RMT}(\text{hide } L_1, \ldots, L_n \text{ in } B_0')) \)
- \( U \text{ abstraction } B_1 \ldots \text{ of } R_{R'M'T'}(B_2') \text{ where } R = R' \rightarrow \mathcal{O}(R_{RMT}(U \text{ abstraction } B_1 \ldots \text{ of } B_2')) \)
- \( \text{otherwise } \rightarrow R_{RMT}(\mathcal{O}(B_0)) \)
6. THE SVL 2.0 COMPILER

A compiler for the SVL language has been implemented within the CADP toolbox. This compiler, named SVL 2.0, includes five phases, as shown in Figure 1. It was developed using an original compiler construction technology also used for two other compilers developed by the VASY team of INRIA:

- The lexical analysis, syntax analysis, and abstract tree construction phases (1,100 lines of code) are implemented using the SYNTAX compiler generator [2]. SYNTAX has similar functionalities as LEX and YACC, enhanced with a well-designed automatic error recovery mechanism.
- Type checking, expansion of meta-operations, and code generation (2,900 lines of code) are implemented as a set of data types and functions written in the LOTOS NT language [16, 26]. Inspired by the standardization work on Enhanced LOTOS [21], LOTOS NT combines the theoretical foundations of process algebras with features borrowed from both functional and imperative languages (such as abstract data types, patterns, assignments, loops, . . . ) suitable for a wider industrial acceptance of formal methods. The TRAIAN compiler [25] translates the LOTOS NT code into a set of C types and functions. The generated C code is augmented with about 200 lines of hand-written C code.

Figure 1. Architecture of SVL 2.0
SVL generates a script written in Bourne shell. This script starts by including a set of predefined functions (1,750 lines of Bourne shell), then the SVL statements and expressions are translated into sequences of calls to these functions with suitable parameters.

In total, the development of SVL 2.0 took about 5 person · month, and totalizes more than 11,000 lines of generated C code. The SVL compiler is designed carefully to ease the verification task. We give five examples of such a careful design:

- When executing a verification scenario, SVL 2.0 produces simultaneously two kinds of output: a high-level, concise execution trace expressed at the abstraction level of the SVL source program, and a low-level log file which lists all the executed shell commands and the output of their execution. This file is very convenient to locate and understand run-time errors.

- During its execution, the generated script produces several temporary files (explicit LTSs, Exp files, hiding and renaming files, etc.) needed by the various tools of CADP. To minimize disk space consumption, SVL removes temporary files as soon as possible and uses as much as possible the BCG format for representing explicit LTSs, because the BCG format is more compact than other LTS formats.

- Several convenient compiler options are implemented, e.g., “–debug”, which prevents intermediate files from being erased, “–expand”, which produces the SVL program obtained by expanding all meta-operations, and “–script”, which generates the Bourne shell script without executing it.

- As much as possible, the generated script tries to recover from errors automatically by using “expert” knowledge about the verification process. For instance:
  - When a reduction fails because the combination of relation, tool, and method specified in the source SVL program is not available, SVL 2.0 attempts to change some of the parameters (tool and/or method). For instance, it can replace “std” with “bdd” when using ALDEBARAN to perform strong reduction of an LTS in the Exp format since standard reduction is not available in this case.
  - If a weak reduction fails, e.g., because of memory exhaustion, SVL 2.0 tries to perform first stronger reduction(s) before trying again the weak reduction. For instance, observational reduction may be preceded by branching reduction, and branching reduction may be preceded by strong reduction. If the reduction still fails, SVL 2.0 will leave the LTS un-reduced and continue execution.
– The S VL semantics states that a “generation” operator is mandatory whenever an implicit LTS should be converted to an explicit LTS because such conversion is costly and can lead to state explosion. Practically, this constraint is slightly relaxed: if a “generation” operator is omitted, it will be inserted automatically and a warning message will be emitted.

- S VL 2.0 permits to invoke shell commands from the S VL description. This can be used for calling tools with features not implemented in S VL, for using the shell control structures to perform conditionals and loops, and for modifying the values of environment variables specifying the default tools, methods, and relations as well as options to be passed to the tools.

7. CONCLUSION

Although compositional verification has a strong potential in attacking the state explosion problem, only a few implementations have been carried out. In this respect, the compositional verification tools of CADP are of particular interest because they are widely distributed and have been applied to several industrial case-studies [4, 22, 24, 19]. However, these tools require a good level of expertise to be effective. To address this problem, we designed the scripting language S VL, which is well-adapted to compositional verification:

- S VL combines process algebra operators (parallel composition, label hiding) with operations on LTSs (e.g., minimization, abstraction, comparison, livelock/deadlock detection, label hiding and label renaming using regular expressions, etc.). It also provides high-level meta-operators, which allow sophisticated compositional verification strategies to be expressed concisely. Practically, S VL is expressive enough to supersede the two formats Exp and Des previously used in the CADP toolbox and to suppress the need for hand-written MAKEFILES.

- S VL behaves as a tool-independent coordination language (in the same way as EUCALYPTUS [12] is a tool-independent graphical user interface). Due to its concise syntax and well-chosen default options, S VL relieves the user from the burden of launching verification tools manually: it invokes the relevant tools with appropriate options and allocates/deletes temporary files automatically. S VL supports several verification tools (ALDÉBARAN, BCG_MIN, FC2TOOLS) both for explicit (enumerative) and implicit (on-the-fly) verification. It supports several LTS formats (AUT, BCG, FC2, SEQ) though using as much as possible the BCG format, which allows significant savings in both disk space and access time. Switch-
changing from one tool or one format to another can be done simply by changing a few words in the S VL description.

- **S VL** is extensible in two ways. Its modular design will allow new tools and new formats to be integrated easily. Moreover, as Bourne shell commands can be directly invoked from an S VL description, the user can easily describe specific processings and benefit from the high-level constructs of the Bourne shell (if and case conditions, while and for loops, etc.).

For this language, we have fully implemented the S VL 2.0 compiler which is developed using an original approach to compiler construction, combining the SYNTAX compiler generator and LOTOS NT, a variant of the forthcoming E-LOTOS standard.

S VL 2.0 has reached a good level of stability and maturity. It is distributed as a component of CADP and available on four different UNIX and WINDOWS platforms. It has been used in ten case-studies profitably, in particular for the compositional verification of a dynamic reconfiguration protocol for agent-based applications [1]. As regards future work, three directions can be identified:

- **S VL** could be enhanced with common sub-expressions detection. At present, the user can always avoid redundant computations by storing their results in intermediate, named files, which can be reused later. For instance, the statement:

\[
"a\cdot bcg" = \text{leaf strong reduction of} \\
((B_1 \mid [G] \mid B_0) \mid (B_2 \mid [G] \mid B_0))
\]

can be evaluated more efficiently as:

\[
"b\cdot bcg" = \text{strong reduction of } B_0 \\
"a\cdot bcg" = ((\text{strong reduction of } B_1) \mid [G] \mid "b\cdot bcg") \mid (\text{strong reduction of } B_2) \mid [G] \mid "b\cdot bcg")
\]

Ideally, this optimization could also be performed automatically.

- The **S VL** language could be enriched with additional operators, e.g., to model-check temporal logic formulas, and additional metac-operators, such as recursive propagation of hiding (so as to hide labels as soon as possible) or recursive abstractions. More applications are needed to determine which extensions are practically useful.

- The **S VL** language and related tools should be extended to support the new parallel composition operators [17] introduced in E-LOTOS and LOTOS NT; these operators are more expressive and user-friendly than LOTOS ones and would thus contribute to make compositional verification easier.
Acknowledgements

We would like to thank the former and current members of the \textsc{Inria/Vasy} team involved in the Svl project, namely: Mark Jørgensen and Christophe Discours, who implemented the early versions (1.0–1.6) of the Svl compiler, Ghassan Chehaibar and Charles Pecheur, who used and gave valuable feedback about the \textsc{Des2Aut} and \textsc{Svl 1.*} tools, Marc Herbert and Stéphane Martin, who tested the improved \textsc{CAdp} tools on several architectures carefully, and Radu Mateescu, who experimented the Svl 2.0 compiler on several case-studies.

We are also grateful to Ji He (University of Stirling) for her reports about the \textsc{Des2Aut} tool, to Laurent Mounier (Univ. Joseph Fourier, Grenoble) for his pertinent explanations regarding \textsc{Des2Aut} in particular and compositional verification in general, and to Solofo Ramangalahy, and to the anonymous referees for their useful comments about this paper.

REFERENCES


ON FORMAL TECHNIQUES
IN PROTOCOL ENGINEERING
— EXAMPLE CHALLENGES

Dines Bjørner
Informatics and Mathematical Modelling, Technical University of Denmark
Building 322, Richard Petersens Plads, DK–2800 Lyngby, Denmark
URI: www.imm.dtu.dk/~db, E-Mail: db@imm.dtu.dk

Abstract  In this invited paper — for the 2001 FORTE Conference on FORmal Description TEchniques for Protocol Specification, Testing and Verification — we intend to illustrate a number of methodological issues that the practising software engineer might wish to consider when developing software for interacting, distributed software, more precisely, for such software which significantly contains interaction protocols. We “attach” our “guided tour” of software engineering using formal techniques, and following a triptych of domain engineering, requirements engineering, and software design, to two example “challenges”. The particular examples are the domain and requirements for an “inverse” smart card application, that is, of a domain where ordinary citizens gather, refer to and update (“change”) accumulated information and money resources and of requirements to a “really” smart card system which is the primary carrier of (ie. repository for) such information, and the (electronic) monies (themselves). The other example is that of “market” transactions; that is, a domain of interactions between government (G) institutions, businesses (B), and citizens (C). That is with $G \leftrightarrow B \leftrightarrow C$ sequences of interactions — ie. “protocols”, as well as some thoughts on requirements (an “extended E-Business system”).

We outline the problems while adhering to our approaches to both informal and formal software engineering, to the development triptych, and indicate some of the very many principles and techniques inherent in any good method. By showing two different examples we illustrate the “impossibility” of one simple, straightforward method.

Thus the paper presents a capsule view of some of our view of the Theory & Practice of Software Engineering.¹

¹This is also the title of: [1].
1. INTRODUCTION

Protocol engineering, a sub-discipline of software engineering, as automotive engineering is of mechanical engineering, is a complex professional field. Protocol designers understood — earlier than software engineers in general — the importance of formal specification and verification.

Perhaps protocol engineers focused more on abstractly specifying intricate protocols and verifying some of their properties, than on developing, in general, provably correct software that satisfied the protocol specifications? This paper is about the latter.

In this presentation we wish to alert the field of protocol engineering to a number of more general software engineering development principles and techniques whose use in other fields of software engineering has been of some use.

Our two main points are those of the software engineer vigilantly following and adhering to a long check list of method principles and techniques, and that protocol development step back to include careful considerations of the actual world, ie. the domain, in which protocols serve, to better understand as many of the issues that ultimately lie behind a great number of protocol properties.

For software engineering to eventually label itself a professional engineering activity it must, it seems, rely, as do other engineering professions, on high specialisation — hence our insistence on referring to protocol engineering. Specialisation is characterised in terms of varieties of principles, techniques and tools whose conditional application follows careful analyses. Our field is slowly accumulating and enunciating these principles, techniques and tools. It may take a whole generation more to get them into general use.

1.1. “Formal” Methods

1.1.1 A Caveat. The label ‘“formal” method’ has been affixed to such techniques as use “formal” specification and “formal” verification. Here we first double quote the term ‘method’, secondly we “reduce” the scope of ‘formal’ from ‘method’, to ‘specification & verification’. Double–quoting expresses that we do not quite mean what we write: Methods cannot be “formal”, but some of the techniques “they” imply may.

²The ‘&’ connective is here used in its conventional meaning: Its two operands with ‘&’ form one concept, not two !
1.1.2 “What is a Method ?”. Here the double quoted slanted display line title text designates a rhetorical question. Our answer is: A method is a set of analysis and synthesis principles for selecting and applying a number of techniques and tools in order efficiently to construct an efficient artifact. Here those artifacts are software.

Principles are to be adhered to, ie. followed, by humans. Hence they cannot be formalised, ie. are not “formal”. Some techniques and tools (addressing such techniques) can be said to be formal. Specification languages are tools, and certainly can be formal: Have formal semantics, based on formal syntax, but never a formal pragmatics: Pragmatics is what leads outside formality: Has to do with the socio–economic context in which any utterance is made. Hence methods cannot be “formal”, but crucial aspects of techniques and hence their associated tools can.

1.2. A Triptych of Software Engineering

Before software can be designed we must understand its requirements, hence they must (first ?) be engineered. Before requirements can be developed we must understand the (application) domain, hence it must (first ?) be engineered. Our approach is to both informally and formally describe all three: Domains, requirements and software — perhaps — in many stages and steps of “refinement”. And to relate these phases, stages and steps.

1.3. Problem Synopses

Engineering is “walking the bridge between science and technology”: “Constructing artifacts (here computing systems, notably software) based on scientific insight,” and “investigating technology in order to ascertain its possible scientific value.”

In this paper, as in so much else, we, engineers, are “driven” (forward, or is it sometimes “backwards”) by the promises of the hard technologies.

In the case of this paper it is the “technologies” of “smart-cards” and “Internet”, and their “derivatives” that first “drive’ us.

1.3.1 The Citizen Smart Card: The Technology Incentive. Imagine, a few years from now, that low power, mass storage, fast computing & optical communication is being facilitated by credit card sized (incl. thin) cards — which we shall henceforth refer to as The Citizen Smart Cards.

The Citizen Smart Card is capable, we conjecture, of holding, typically all of a citizens accumulated health–care information, and more, from a full lifetime: Information about all the health states, temperatures,
blood pressure, blood sugar, cholesterol content, etc., all the original ECGs ever measured, all the original X–Rays ever taken, special photos taken before and after cosmetic (plastic) surgery, all the patient medical records ever recorded during hospital and clinic visits, your family doctors’ notes about your health. All of this time stamped, etc.; all of a citizens accumulated financial information, again for any full lifetime, as well as all financial instruments currently holdable by any such citizen — ie. the monies themselves suitable transport timetables as well as commuter (daily, weekly, monthly or seasonal) tickets.

Add to the above that the first two kinds of information held on The Citizen Smart Card, in principle, is held nowhere else: No copies!³ The citizen is in sole and full possession of all own information!

The development brief now being posed, as our first example, is: Now what does the above imply in a world where we forget this card, where we loose it, where card terminals break down, etc.? Which citizen/card/machine interaction protocols does this call for?

We will try “edge” towards an answer by examining and modelling, also formally, the actual world of these three information source in the actual domain, and then suggest properties that a protocol and its supporting machinery required to satisfy.

1.3.2 E–Business: The Technology Incentive. The Internet is here. Here and there, contractual transactions take place over the net. Merchandise (ie manifest, tangible goods, instantly tradeable, transferable) and (intangible, time–interval providable) services are offered, are sold, are sought, are bought, etc.

So far two categories of traders: Buyers and sellers have been identified: Businesses (B) and citizens (or consumers, C). But in general there are three, the above and government institutions (G).

The question is now: How to generalise the notions of B2C and B2B transactions to arbitrary pairings of traders (buyers and sellers) from the three categories; and how to extend these transactions to such which involve “middlemen”: Agents and brokers?

The development brief now being posed, as our second example, is — in the domain — to investigate the full triangle of pairwise interactions; identify “primitive” such transactions, such which enter into most transaction sequences; explore possible behaviours; and — as requirements — to identify suitable protocols.

³That is: No copies, in a logical sense. Physically, instead of The Citizen Smart Card storing bulks of information it may instead refer to more stationary repositories where (sole) originals are kept. Still, the idea is that the citizen is in full control.
1.4. Structure of Paper

In Section 2 we informally and formally describe the two domains of the citizen information “repository”, and “the market”. Based on that, we can, in Section 3, outline the Citizen Smart Card and E-Business requirements — emphasising their protocol aspects. In those sections, and in Section 4, we review a number of method principles and techniques. Space limitations do not permit a thorough treatment — so we shall refer, instead, to published papers and a forthcoming text book on methodical software engineering based on the use of formal techniques.

2. DOMAIN ENGINEERING

Domain engineering is not the same as knowledge engineering. The latter could be said to be “a subset of” the former. In knowledge engineering, briefly, computing scientists and knowledge engineers try model the knowledge of stakeholders of the domain in such a way that the knowledge models are “directly” computable. In domain engineering we do not consider computable aspects only. We wish to model as much of the domain that can be described. We refer to [2] for details.

2.1. The Citizen Smart Card: A Domain

2.1.1 Three Sub-Domains: Informal Sketches. We treat three diverse information and resource “gathering” and “dispensing” domains: citizen health information, financial information and securities, and transport services and tickets.

Patient Medical Records (PMRs) and Journals (PMJs) By a Patient Medical Record, PMR, we mean a collection of documents pertaining to one particular health case of one particular citizen — whether that citizen sought help from medical professionals or not. At any one time several PMRs may be in effect: The citizen may be analysed or undergoing treatments for more than one disease or similar. By a Patient Medical Journal, PMJ, we mean the collection of all PMRs, over the entire life-time of that patient, “up till now!”.

We refer to PMRs, not necessarily as manifest, tangible paper, instrument output strips or X-ray files, documents, but, in addition to such, also as what the citizens (the patient, relatives working place colleagues), in the past remembered, or now remembers. We certainly do not have electronic PMRs in mind but do not a-priori rule them out either.
In other words: A PMJ represents all that is knowable, by whatever means, about a citizen’s health.

A PMJ may be subject to a number of operations: (1) creation, ie. opening of new PMRs, and (2) closing, ie. “filing away” of existing PMRs. Individual PMRs may be subject to a number of operations: (3) entering and update of administrative information: Patient name, dates, addresses, etc.; (4) entering and update, incl. “deletion” of “raw”, undigested (ie., further unanalysed) medical information obtained from the patient, “previous” PMRs, relatives, etc., or obtained as analysis results from medical professionals (or other), or in the form of analysis originals, or from, analytical (medico-technical) instruments (ECGs, blood analysis, X—Rays, etc.); (5) entering and update, incl. “deletion” of diagnostics; (6) entering and update of treatment plans; (7) entering and update of treatment actions; (8) entering and update of how the patient responds to treatment actions. Whereby we are back to item (4).

In addition we can inquire about PMR/PMJ information: search, correlate, and compute over such information — as do medical professionals: When analysing, when diagnosing, etc.

That’s basically “all!”

We refer to two kinds of operations and of two kinds of queries: those which update PMJs with no input provided (O), those which update PMJs with input provided (IO), those which query PMJs with no input provided (Q), and those which update PMJs with input provided (IQ).

Financial Services We treat only banking and securities trading.

- **Banking** Today you would be expected to have a number of bank accounts (Acc): Savings & Loan (S&L), Demand/Deposit, Cheque, etc. Each of these show a balance, either positive or negative. Each such balance represents **credit** or **debit** in the bank. One may perform the following operations related to accounts: open, close (new, resp. old account), save, borrow, repay (in preparation for loan, establish loan, repay loan — in installments), withdraw, deposit, change credit (latter stands for extending or lowering credit limit), statement (list of transactions “since last”), etc.

The development brief calls for an analysis of the bank/customer interaction interface today — with a view towards keeping the banks, but maintaining all information and all balances (whether negative or positive) — that is: “the real monies” — on The Citizen Smart Card in future.
● Securities Trading  Securities trading is an important financial service: Enterprises and governments offer securities (stocks, respectively bonds, Sec), for sale. Enterprises and people buy and sell stocks and bonds. Stocks and bonds are traded on the “floor” of stock exchanges. Brokers and traders act as middlemen between buyers and sellers. The following are typical securities trading operations: initial stock offering; placing and withdrawing orders to buy, respectively sell; effect, as it were “simultaneously” a batch of one or more buy/sell orders; transfer of securities instruments: The stock, bonds and payments. &c.

The development brief calls for an analysis of the client/broker interaction interface today — with a view towards keeping the broker and securities exchanges, but maintaining all information and all securities instruments (stocks, bonds) on The Citizen Smart Card in future.

Public Transport  We shall look at Public Transport only from the very narrow perspective of potential passengers (incl. commuters). Commuters possess knowledge of time and fare tables. Commuters enter, pay for (cash or by showing seasonal tickets), and leave transport vehicles (taxis, busses, street cars, metro trains, canal boats, etc.).

The development brief calls for an analysis of the commuter/transport system interaction interface today — with a view towards keeping the transport, but maintaining all information (time and fare tables), and all monies and tickets (InfoTick) on The Citizen Smart Card in future.

2.1.2 Analysis and Formal Model. We decompose our analysis and models into data (ie. type) and function (incl. process) models.

Data Analysis  The unifying quantity, whether PMR, PMJ, Sec, or InfoTick, will be referred to as \( R \). Operations on (ie. functions that change) PMR, PMJ, Sec, or InfoTick, will be referred to a \( O \). Queries on PMR, PMJ, Sec, or InfoTick, will be referred to a \( Q \). Extensions of these are operations or queries which update with special information or query wrt. such information \( I \).

Formal Domain Data Model

\[
\begin{align*}
type \quad & R, I \\
O \quad & R \rightarrow R \\
Q \quad & R \rightarrow I \\
IO \quad & I \times R \rightarrow R \\
IQ \quad & I \times R \rightarrow I
\end{align*}
\]
Process Analysis  Whether a PMR or a PMJ: moving around the health-care system, being tested, taking medicine, having a fever, etc.; or a bank account or a securities instrument (Acc, Sec): depositing and withdrawing cash into and from an account (etc.), buying or selling stocks and bonds (etc.), etc.; or a transport repository (Info Tick): obtaining new time table and fare information, boarding and leaving a bus (train, etc.); all amounts to updating or querying the repository.

Formal Process Model  Each repository is like a process. Each such process is, non–deterministically, subject to any of a great variety of operations or queries.

\[
\begin{align*}
type \\
\text{Idx} &= \{ 1..n \} \\
\Theta &= \text{Idx} \rightarrow R \\
value \\
\theta : \Theta \\
\text{system}(n) &\equiv \{ \text{citizen}(\theta(i)) | i: \text{Idx} \} \\
\text{citizen}(r) &\equiv \\
&\quad \text{let } oq = \text{io}, o, iq, q \text{ in} \\
&\quad \text{cases } oq \text{ of} \\
&\quad \quad \text{io } \rightarrow \text{let } \text{io:IO}, i:i \text{ in } \text{citizen}(\text{io}(i,r)) \text{ end} \\
&\quad \quad o \rightarrow \text{let } \text{o:O} \text{ in } \text{citizen}(\text{o}(r)) \text{ end} \\
&\quad \quad iq \rightarrow \text{let } \text{iq:IQ}, i:i \text{ in } \text{iq}(i,r) \text{ end} \\
&\quad \quad q \rightarrow \text{let } q:Q \text{ in } q(r) \text{ end} \\
&\quad \text{end end}
\end{align*}
\]

The system of all \((n)\) citizens proceed, in the domain, as a set of non–interfering parallel processes. Each citizen non–deterministically (internal choice, \(\rightarrow\)) perform an information–based update (io), a “standard” (a simple) update (o), an information–based query (iq), or a (simple) query (q).

To paraphrase: That’s all !

Discussion  We remind the reader that the narration and formalisation above is of a domain, not of requirements to software. And we remind the reader that we are focusing on the repository of information and certain (here the financial) resources.

In this example the formal domain model is very trivial. It expresses only the very basics. In the domain all is possible: Humans err, are not consistent, etc. One reason for seeking computing and communication support is to help avoid mistakes and to secure consistency.
The protocol issue was highlighted: We have modelled the non-deterministic handling, i.e. arbitrary interleaving of the operations and queries of this information and resource repository. We have abstracted which these operations and queries were in specific.

The next example will be in contrast to the present.

2.2. E–Business: A Domain

2.2.1 Government, Business and Citizen Transactions. In the case of the government, business and citizen infrastructure interactions we can postulate the following domain, i.e. not necessarily computer & electronic communication supported, transactions:

\[ G2G: \text{Government institutions, } G, \text{ buy services from other } G, \text{ and } G \text{ sell services to other } G. \text{ The } G \text{ pay with monies (obtained through taxes, etc.), respectively offer free services, in return. } G2B: G \text{ buy services, or request taxes, from businesses (} B, \text{ and pay, respectively offer free services, in return. } G2C: G \text{ buy services (hire), or request taxes, from citizens (} C, \text{ and pay, respectively offer free services, in return. } B2G: B \text{ buy services from } G, \text{ and pay } B \text{ for these either through having already paid taxes or by paying special fees. } B2B: B \text{ buy merchandise or services from other } B, \text{ and } B \text{ offer merchandise or services to other } B. B \text{ usually pay for these outright. } B2C: B \text{ buy services from citizens: i.e. hire temporary or permanent staff (employment), and } B \text{ pay for these through salaries. } C2G: C \text{ obtain services from } G \text{ (passport, drivers licence, etc., health-care, education, safety and security, etc.) and } C \text{ pay for these either by paying special fees or through having already paid taxes. } C2B: C \text{ buy merchandise from } B, \text{ and } C \text{ pay for this. } C2C: \text{ Two or more } C \text{ together enter into political “grass–root” organisations, or leisure–time hobby club activities, or just plainly arrange meetings (incl. BBQ parties); and the two or more } C \text{ “pay” for this by being “together”.}

2.2.2 Traders: Buyers and Sellers. Above we have stressed that also government (institutions) are part of the more general concept of E–Business, some aspects of contractual obligations, and a seeming “symmetry” between partners to any such contract (i.e. buy, sell, etc.). As such we have stressed that “The Market” consists of buyers and sellers, whom we, as one, refer to as traders.

2.2.3 Traders: Agents and Brokers. An agent, to us, while we are still only discussing the domain, is a trader that acts (in a biased manner) on behalf of usually one other trader (either a buyer, or a seller), vis–a–vis a number of other traders (sellers, respectively buyers), in
order to secure a “best deal”. A broker, to us, while we are still only discussing the domain, is a trader that acts (in a neutral manner) on behalf one or more buyers and one or more sellers in order to help them negotiate a “deal.”

2.2.4 Schematic Transactions. Sequences of contractual transactions can be understood in terms of “primitive” transactions:

A buyer inquires as to some merchandise or service. A seller may respond with a quote. A buyer may order some merchandise or service. A seller may confirm an order. A seller may deliver an order. A buyer may accept a delivery. A seller may send an invoice. A buyer may pay according to the invoice. A buyer may return, within warranty period, a delivery. And a seller may refund such a return.

We have, deliberately, used the “hedge” ‘may’:

A trader may choose an action of no response, or a trader may inform that a transaction was misdirected, or a trader may decline to quote, order, confirm, deliver, accept, pay or refund!

2.2.5 Formalisation of Syntax.

\[
\text{type} \\
\text{Trans} \equiv \text{Inq} | \text{Ord} | \text{Acc} | \text{Pay} | \text{Rej} \\
\text{Qou} | \text{Con} | \text{Del} | \text{Acc} | \text{Inv} | \text{Ref} \\
\text{NoR} | \text{Dec} | \text{Mis}
\]

The first two lines list the ‘buyer’, respectively the ‘seller’ initiated transaction types. The third line lists common transaction types.

U below stand for unique identifications, including time stamps (T), Sui for surrogate information, and MQP alludes to merchandise identification, quantity, price.

\[
\text{U, T, Su1, Su2, MQP} \\
\text{Inq} = \text{MQP} \times \text{U} \\
\text{Qou} = (\text{Inq} | \text{Su1}) \times \text{Inf} \times \text{U} \\
\text{Ord} = \text{Qou} | \text{Su2} \times \text{U} \\
\text{Con} = \text{Ord} \times \text{U} \\
\text{Del} = \text{Ord} \times \text{U} \\
\text{Acc} = \text{Del} \times \text{U} \\
\text{Inv} = \text{Del} \times \text{U} \\
\text{Pay} = \text{Inv} \times \text{U} \\
\text{Rej} = \text{Del} \times \text{U} \\
\text{Ref} = \text{Pay} \times \text{U} \\
\text{NoR} = \text{Trans} \times \text{U} \\
\text{Dec} = \text{Trans} \times \text{U}
\]
$$\text{Mis} = \text{Trans} \times U$$

value

$$\text{obs}_T : U \rightarrow T$$

In general we model, in the domain, a “subsequent” transaction by referring to a complete trace of unique, time stamped transactions. Thus, in general, a transaction “embodies” the transaction it is a manifest response to, and time of response.

Figure 1 attempts to illustrate possible transaction transitions between buyers and sellers.

2.2.6 Transaction Sequences. Figure 3 attempts to show the possible sequences of transactions as related to one “trade”: From inquiry through refunding, that is: For one pair of buyer/seller.

2.2.7 “The Market”. Figure 2 attempts to show that a trader can be both a buyer and a seller. Thus traders “alternate” between buying and selling, that is: Between performing ‘buy’ and performing ‘sell’ transactions.

Figure 4 attempts to show “an arbitrary” constellation of buyer and seller traders. It highlights three supply chains. Each chain, in this example, consists, in this example, of a “consumer”, a retailer, a wholesaler, and a producer.

2.2.8 Formalisation of Process Protocols. “The Market” consist of $n$ traders, whether buyers, or sellers, or both; whether additionally agents or brokers. Each trader $\tau_i$ is able, potentially to com-
municate with any other trader \( \{ \tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n \} \). We omit treatment of how traders come to know of one another. And focus only on the internal and external non–determinism which is always there, in the domain, when transactions are selected, sent and received.

Our model is in a variant of CSP, but expressed “within” RSL [3].

\[
\text{type } \Theta = \{ \mid 1..n \mid \}
\]

\[
\text{value}
\]
Figure 4. A Network of Traders and Supply Chains

sys: (Idx \ \Theta \ ) \times n : \textbf{Nat} \rightarrow \textbf{Unit}

sys(m\theta,n) \equiv \big\| \{ \text{tra}(i)(m\theta(i)) \mid i : \text{Idx} \} \big\|

tra: i : \text{Idx} \rightarrow \Theta \rightarrow

\text{in} \ \{ tc[j,i] \mid j : \text{Idx} \cdot i \neq j \} \textbf{out} \ \{ tc[i,j] \mid j : \text{Idx} \cdot i \neq j \} \textbf{Unit}

tra(i) (\theta) \equiv \text{tra}(i)(\text{nxt}(i)(\theta))

nxt: i : \text{Idx} \rightarrow \Theta \rightarrow

\text{in} \ \{ tc[i,j] \mid j : \text{Idx} \cdot i \neq j \} \ 
\text{out} \ \{ tc[i,j] \mid j : \text{Idx} \cdot i \neq j \} \ 

\Theta

\text{let} \ \text{choice} = \text{rcv} \mid \text{snd} \ 
\text{in} \ 
\text{cases} \ \text{choice} \ \text{of}

\text{rcv} \rightarrow \text{receive}(i)(\theta), \ \text{snd} \rightarrow \text{send}(i)(\theta) \ \text{end} \ \text{end}

The system is the parallel combination of n traders. Traders communicate over channels: tc[i,j] — from trader i to trader j. Each trader is modelled as a process which "goes on forever", but in steps of next state transitions. The next state transition non—deterministically (internal choice, \{\}) "alters" between expressing willingness to receive, respectively desire to send. In "real life", ie. in the domain, the choice as to which transactions are taken is non—deterministic. And it is an internal choice. That is: The choice is not influenced by the environment.

receive: i : \text{Idx} \rightarrow \Theta \rightarrow \text{in} \ \{ tc[j,i] \mid j : \text{Idx} \cdot i \neq j \} \ 

receive(i)(\theta) \equiv

\text{let} \ 
\text{msg} = \text{tc[j,i]}? \ 
\text{in} \ 
\text{update}_\text{rcv}_\text{state}(\text{msg},j)(\theta) \ \text{end} \mid j : \text{Idx} \}

update\_\text{rcv}_\text{state}: \text{Idx} \times \Theta \rightarrow \Theta
*update_rcv_state* is not a protocol function. *update_rcv_state* describes the deposit of *msg* in a repository of received messages. If *msg* is a response to an earlier sent transaction, *msg-o*, then *update_rcv_state* describes the removal of *msg-o* from a repository of sent messages. *remove_set_msg* models the situation where no response (*nor*) is (ever) made to an earlier sent message. Once the internal non–deterministic choice ([[]]) has been made: Whether to *receive* or *send*, the *choice* as to whom to ‘receive from’ is also non–deterministic, but now external ([[]]). That is: *receive* expresses willingness to receive from any other *trader*. But just one. As long as no other *trader j* does not send anything to *trader i* that *trader i* just “sits” there, “waiting” — potentially forever. This is indeed a model of the real world, the *domain*. A subsequent *requirement* may therefore, naturally, be to provide some form of *time out*. A re–specification of *receive* with time out is a *correct implementation* of the above.

\[
\text{send: } i: \text{Idx} \rightarrow \Theta \rightarrow \text{in} \{ \text{tc}[i,j]| j: \text{Idx} \neq j \} \Theta \\
\text{send (i)(} \Theta \equiv \\
\text{let choice = ini [] res [] nor in} \\
\text{cases choice of} \\
\text{ini } \rightarrow \text{send_initial (i)(} \Theta \)), \\
\text{res } \rightarrow \text{send_response (i)(} \Theta \)), \\
\text{nor } \rightarrow \text{remove_received_msg (} \Theta \end{end} 
\]

Either a *trader*, when communicating a transaction chooses an *initial (ini)* one, or chooses one which is in *response (res)* to a message received earlier, or chooses to *not respond (nor)* to such an earlier message The choice is again non–deterministic internal. In the last case the state is updated by *non–deterministically internal choice* (not shown) removing the, or an earlier *received message*.  

Note that the above functions describe the internal as well as the external non–determinism of protocols. We omit the detailed description of those functions which can be claimed to not be proper protocol description functions — but are functions which describe more the particular *domain* at hand: Here “The Market”.

\[
\text{send_initial: } i: \text{Idx} \rightarrow \Theta \rightarrow \text{out} \{ \text{tc}[i,j]| j: \text{Idx} \neq j \} \Theta \\
\text{send_initial (i)(} \Theta \equiv \\
\text{let choice = buy [] sell in} \\
\text{cases choice of} \\
\text{buy } \rightarrow \text{send_int_buy (i)(} \Theta \)), \\
\text{send } \rightarrow \text{send_init_sell (i)(} \Theta \end{end} 
\]
send_response: $i: \text{Idx} \rightarrow \Theta \rightarrow \text{out}$ \quad \{tc[i,j] \mid j: \text{Idx} \cdot i \neq j\} \Theta
send_response(i)(\theta) \equiv
  \begin{align*}
  &\text{let choice} = \text{buy} \mid \text{sell} \quad \text{in} \\
  &\text{cases choice of} \\
  &\quad \text{buy} \rightarrow \text{send_res_buy(i)(}\theta) \\
  &\quad \text{sell} \rightarrow \text{send_res_sell(i)(}\theta) \quad \text{end end}
\end{align*}

In the above functions we have, perhaps arbitrarily chosen, to distinguish between buy and sell transactions. Both send_initial and send_response functions — as well as the four auxiliary functions they invoke — describe aspects of the protocol.

send_init_buy: $i: \text{Idx} \rightarrow \Theta \rightarrow \text{out}$ \quad \{tc[i,j] \mid j: \text{Idx} \cdot i \neq j\} \Theta
send_init_buy(i)(\theta) \equiv
  \begin{align*}
  &\text{let choice} = \text{inq} \mid \text{ord} \mid \text{pay} \mid \text{ret} \mid \ldots \quad \text{in} \\
  &\text{let} (j, \text{msg}, \theta') = \text{prepare_init_buy(choice)}(i)(\theta) \quad \text{in} \\
  &\text{tc}[i,j]!\text{msg} ; \quad \theta' \quad \text{end end}
\end{align*}

send_init_sell: $i: \text{Idx} \rightarrow \Theta \rightarrow \text{out}$ \quad \{tc[i,j] \mid j: \text{Idx} \cdot i \neq j\} \Theta
send_init_sell(i)(\theta) \equiv
  \begin{align*}
  &\text{let (choice)} = \text{quo} \mid \text{con} \mid \text{del} \mid \text{inv} \mid \ldots \quad \text{in} \\
  &\text{let} (j, \text{msg}, \theta') = \text{prepare_init_sell(choice)}(i)(\theta) \quad \text{in} \\
  &\text{tc}[i,j]!\text{msg} ; \quad \theta' \quad \text{end end}
\end{align*}

prepare_init_buy is not a protocol function, nor is prepare_init_sell. They both assemble an initial buy, respectively sell message, \text{msg}, a target trader, \text{j}, and update a send repository state component.

send_res_buy: $i: \text{Idx} \rightarrow \Theta \rightarrow \text{out}$ \quad \{tc[i,j] \mid j: \text{Idx} \cdot i \neq j\} \Theta
send_res_buy(i)(\theta) \equiv
  \begin{align*}
  &\text{let} (\theta', \text{msg}) = \text{sel_update_buy_state}(\theta), \\
  &\quad j=\text{obs-trader}(\text{msg}) \quad \text{in} \\
  &\text{let} (\theta'', \text{msg}) = \text{response_buy_msg}(\text{msg})(\theta') \quad \text{in} \\
  &\text{tc}[i,j]!\text{msg}'; \quad \theta'' \quad \text{end end}
\end{align*}

send_res_sell: $i: \text{Idx} \rightarrow \Theta \rightarrow \text{out}$ \quad \{tc[i,j] \mid j: \text{Idx} \cdot i \neq j\} \Theta
send_res_sell(i)(\theta) \equiv
  \begin{align*}
  &\text{let} (\theta', \text{msg}) = \text{sel_update_sell_state}(\theta'), \\
  &\quad j=\text{obs-trader}(\text{msg}) \quad \text{in} \\
  &\text{let} (\theta'', \text{msg'}) = \text{response_sell_msg}(\text{msg})(\theta') \quad \text{in} \\
  &\text{tc}[i,j]!\text{msg}'; \quad \theta'' \quad \text{end end}
\end{align*}

sel_update_buy_state is not a protocol function, neither is sel_update_sell_state. They both describe the selection of a previously deposited,
buy, respectively a sell message, \(msg\), (from it) the index, \(j\), of the trader originating that message, and describes the update of a received messages repository state component. \(\text{response\_sell\_msg}\) and \(\text{response\_buy\_msg}\) both effect the assembly, from \(msg\), of suitable response messages, \(msg'\). As such they are partly protocol functions. Thus, if \(msg\) was an \textit{inquiry} then \(msg'\) may be either a \textit{quote}, \textit{decline}, or a \textit{misdirected} transaction message. Etcetera.

\subsection*{2.2.9 Discussion.} In this example, in contrast to the E–Business example, the protocol aspect was quite “pronounced”. Again we remind the reader that we have, so far only described facets of the domain. Next we shall deal with requirements.

\section*{2.3. Discussion: Domain Engineering}

The two example cases differ widely: The Citizen Smart Card “resulted” in a “small” domain model. The E–Business model, in contrast, was quite substantive. The latter included quite an elaborate, yet, in essence “free” protocol: Any sequence of transactions will occur in the domain, and there were many variants. The former seems, in the domain, to have an utterly uninteresting “protocol” — if such a term could indeed be used!

\section*{3. REQUIREMENTS ENGINEERING}

We illustrate a novel technique of “deriving” significant features of requirements from the domain description. The idea is that the domain descriptions (and ours were necessarily rather cursory) “point” to the necessity of certain domain requirements. Interface and machine requirements will only be cursorily covered. We refer to [4, 5, 6] and the forthcoming [1] for details.

\subsection*{3.1. Domain Requirements}

Domain requirements “construction” proceeds, it is suggested, by deploying the following techniques: \textit{Projection} — where only those parts of the domain model which is of interest are “carried” forward to requirements. \textit{Instantiation} — where abstract types and functions defined only through their signatures are made more specific. \textit{Extension} — where new domain possibilities, afforded through computing and communication, but hitherto not feasible without, are “added” to the requirements (although “really” a domain issue). \textit{Initialisation} — where the whole issue of how to initialise the context and state configuration components
is dealt with. I.e., the input, vetting and update of configuration data. We shall mostly be concerned with projection and instantiation.

3.1.1 The Citizen Smart Card.

**Projection Synopsis** We project information and securities (incl. money) resources onto The Citizen Smart Card as mentioned earlier. Figure 5 attempts to show one possible rendition of The Citizen Smart Card. The Citizen Smart Card is to store all information and be an electronic purse.

![Figure 5. A Smart Card](image)

**Instantiation Synopsis** We omit consideration of how the repository information and electronic (purse) securities, i.e. R, are represented.

**Extension Synopsis** Here is where some considerations of machine requirements (notably dependability issues) are to be made first. We refer forward to the machine requirements for the The Citizen Smart Card.

We extend as now outlined: To complement The Citizen Smart Card there must be means of “feeding” (health–care, financial and transport services) information onto the card and “moving” electronic money to and from the card. Figure 6 attempts to show two possible such means: An ETM/ATM–like machine, as shown to the left, and a “gadget” as indicated to the right. The ETM/ATM–like machines are well–known. The “gadgets” provide the means of writing and reading information to screens and from for example medico–electronics instruments, and bus and train ticketing gadgets.

Over time the citizen applies the card to a variety of such machines and gadgets (i.e. terminals, SCM). Figure 7 attempts to show, as temporally determined, such a sequence of terminals: The Citizen Smart Card was first issued by SCMa, and last used in connection with SCMn.
In-between The Citizen Smart Card was “applied” to SCMb, . . . , SCMj, SCMk, . . . , and SCMm.

Ignore, for the moment, the arrows between SCMs.

But observe that the trace of SCM usage portrays a virtual, a logic usage: Two or more SCMx and SCMy (where x and y in a sense stands for temporal order), may be the same physical terminal. Thus there may be loops. In fact: They all may be the same terminal. The card holder is "provincial", never gets around!

![Figure 7. Trace of Terminal Use](image)

**Failures of Operation** We can now discuss some possibilities: (i) The Citizen Smart Card may be lost. (ii) The Citizen Smart Card may be stolen. (iii) The smart card terminals may break down.

We discuss possible protocols to be observed for each of these three situations.

(i) The Citizen Smart Card may be lost. We assume each card to have a unique identifier. We assume each SCM to remember all the uniquely time-stamped and card identified transactions it has been engaged in for all applied cards. By, for example, letting The Citizen Smart Card remember the identity of the SCM it was last applied to, a link can be made from the SCM it is currently applied to to that “previous” SCM, which, in turn, can then provide a link to the “current” SCM. If the holder of the The Citizen Smart Card can remember just one of the "past" SCMs, then a new card can be issued with all the accumulated and processed information and securities instruments as per the day of loosing the card. (ii) The Citizen Smart Card may be stolen. By providing, as indicated by Figure 5, a triple identification means: Eye iris “stamp”, finger print, and voice print, any high degree of privacy can be obtained. A stolen card otherwise leads to the same re-establishment protocol as a lost card. (iii) The smart card terminals may break down. Figure 8 attempts to show a failing terminal SCMj. By further providing “leap frog” links, forward and backward, skipping one, as shown, two and/or more SCMs, together with copying also “neighbouring” transactions, a suitable redundancies can, again, help ensure recovery of data. And so forth.
Now recall that the trace of terminals used by a card may use the same SCM repeatedly, or always! This could call, in general, for a set of SCM "buddies": SCMs that are physically different from the ("master") SCM of which they are "buddies". Now, instead of inserting links and copying transaction data to "master" SCMs — when the same as "current" SCM — they are copied, for example, to a rotating selection of "buddies". And so forth!

![Diagram of terminal use](image)

**Figure 8.** "Broken" Trace of Terminal use

**Formalisation of Card/Terminal Interaction** We leave out a formal model of the interlocking process of a card, mutually exclusively, interacting with a terminal.

**3.1.2 E–Business.**

**Projection Synopsis** We omit consideration of all operations and queries, i.e. of any specific o, io, q and iq in respectively O, IO, Q and IQ. and focus only on the communication between traders. We basically ignore the "content" of any transaction, and shall instead focus on automating certain sequences of transactions.

**Instantiation Synopsis** Whereas the domain model of traders was a model, essentially, intrinsically, of human operations, we now try to automate as much as possible the response to received transactions. Thus, as an example: (1) If a consumer order can be delivered by the retailer, without human (retailer staff) intervention, it will be done so. (2) If a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who can deliver — both atomic transactions without human (retailer and wholesaler staff) intervention, it will be done so. (3) And if a consumer order cannot be delivered by the retailer, but that retailer can re-order from a wholesaler, who then cannot deliver, but must re–order from producer, who can deliver — all atomic transactions without human (retailer, wholesaler and producer staff) intervention, it will be done so.
Figure 9 attempts to show the three cases listed above. There might be delays, waiting times, between order receipt and delivery and/or re-ordering, across the supply–chain.

**Extension Synopsis** We introduce electronic traders and brokers. They permit arbitrarily wide inquiries: Potentially to all providers (retailers, wholesalers, or producers) of specified merchandise (or services), offers (“confirmations”) of merchandise (or services) to all “takers” (consumers, retailers, or wholesalers), first–come–first serve (“auction”–like) orders, etc. These roughly sketched domain requirements are considered extensions as they might not be humanly feasible in actual domains.

![Three Supply-chain Cases: (c1) Direct, (c2) via Retailer, (c3) all the way from Producer](image)

**Initialisation Synopsis** Due to our projection we need only consider how traders, agents and brokers initially, and in an ongoing way, come to know of one another. We omit details — “left as an exercise”.

**3.1.3 Discussion: Domain Requirements.** We see a big difference in the two example cases: The **The Citizen Smart Card** case requires consideration of — as we shall later see — fairly, technologically detailed machine requirements, and in reality, as we shall see when all three kinds of requirements have been summarised, require quite an elaborate set of protocols. We are “close” to the hard technologies, and the protocol very much depends on hard technology dependability issues. The **E–Business** case is — most likely — to be implemented “on top” of Internet and the web; hence can rely on its underlying protocols; but requires, as we shall see, a mostly domain requirements–determined protocol.
3.2. Interface Requirements

Interface requirements deal with “phenomena shared between users and the machine” (software + hardware). Interface requirements thus include consideration of such issues as graphical user interfaces (GUIs), input/output in general, and user/machine dialogue sequences.

3.2.1 The Citizen Smart Card. The interface requirements have, in this case, to do with the interface between the card and its user with respect to for example: (i) Authentication: For example sequence or simultaneity of pressing card with designated finger for its ‘print’ recognition, uttering a certain phonetic sequence for “voice print” recognition, while aligning, say a left eye with a minute whole through the card, to see light through it, for an “iris print”. (ii) Transaction Processing: For example which card and terminal “buttons/icons” etc. to press/click in order to initiate, conduct and terminate update and query actions. \( \& \& \). Since they only have marginal relevance to protocol engineering we omit further consideration.

3.2.2 E–Business. As for The Citizen Smart Card we omit detailed consideration of trader interfaces to the electronic market support. This is "classic" web–design!

3.2.3 Discussion: Interface Requirements. The The Citizen Smart Card case raises the most non–standard questions when it comes to interface requirements — it is so hard technology, so “gadget” biased. The E–Business case, in contrast, “falls” within the, by now, “classical” field of web and Internet design.

3.3. Machine Requirements

Machine requirements has to do with determining and describing such issues as (i) performance (temporal and spatial [response times, storage and other tangible resource consumption]), (ii) dependability (availability [up time], accessability [fairness], security [authentication, etc.], fault tolerance, etc.), (iii) maintainability (perfective, adaptive and corrective), (iv) portability (hardware and software platforms: development, execution and maintenance), (v) documentation, \( \& \& \).c.

3.3.1 The Citizen Smart Card. This case, of the two being treated, raises the most issues concerning machine, and in particular dependability requirements. They include: Availability: The up–time of the system of all smart card terminals. This issue has little to do
with the concerns of protocol engineering — so we skip it. **Accessibility:** Provision — in the context of multiple users — of fair access to central resources. Since the entire idea of The Citizen Smart Card is “inverted” storage and computing the issue is almost mute. Anyway this issue also has little to do with the concerns of protocol engineering — so we skip it. **Security:** Ensuring protection of privacy of information and private ownership of securities instruments — in this case — leads to for example the triplet schema of eye iris, voice and finger print. There are some, however scant protocol issues related to this, but we skip them! **Fault Tolerance:** Provision of "soft" failures: That is, piecemeal, initially negligible degradation of service in case of lost or stolen cards and in case of terminal breakdowns. We have already, in the domain requirements section, had to presuppose a possibly elaborate scheme for protection against failures of these three kinds. They are the ones that makes, in our view, for an interesting set of protocols, three of which have been mentioned: The protocol(s) of "standard" (failure–free) operation, the protocol(s) for restoring lost or stolen cards, and the protocol(s) for routing around erroneous terminals. 

3.3.2 **E–Business.** The availability, accessibility, and security requirements are assumed “taken care” of by an “underlying” Internet system. We focus just on the fault tolerance issue: If a trader “goes out of business”, or electronically “breaks down”, while many transactions, from and to many other traders are pending, then what? Here, for example, the possible simplicity of a supply chain protocol, as indicated earlier, is at stake. A proper protocol for handling this requires back-ups, duplication, "proxies" and the like. In other words: Protocol engineering “takes over”!

3.3.3 **Discussion: Machine Requirements.** We have seen that machine requirements could not be kept strictly separate from domain and interface requirements; and we have seen the diversity and differing intensity with which various machine requirements issues “flavours” protocol considerations. Finally we have seen that some protocol aspects, as dictated for example by machine requirements, can be taken care of by assumed implementation platforms, while others lead to additional protocols operating, as it were, “on top of platform protocols”.

3.4. **Challenges**

There are two challenges. They can only be formulated now, after all aspects of the requirements have been listed: Domain, interface and machine requirements.
3.4.1 Challenge I: The Citizen Smart Card Protocols. We challenge the reader to augment the The Citizen Smart Card synopses, and to informally and formally describe the protocols for handling (i) normal card/machine operation (insertion of forward and backward links and the copying of transaction data); (ii) re—establishment of a “new” card, when one is lost or stolen; and (iii) detection of and “rerouting” around failing machines.

3.4.2 Challenge II: E–Business Protocols. As for Challenge I we challenge the reader to informally and formally complete the E–Business synopses, and in particular to informally and formally describe the implied E–Business protocols.

3.5. Discussion: Requirements

The various discussion sections, in connection with domain requirements, interface requirements, and machine requirements, seems to have exhausted the issue. Let us, however, just remind the reader that one example’s “thin” domain description “explodes” into a hefty protocol requirements description, while another example’s “substantial” domain description “fizzles” into a slightly augmented protocol requirements description, We refer to (one) The Citizen Smart Card versus (another) E–Business.

4. FURTHER METHODOLOGY ISSUES

4.1. Abstraction & Modelling Principles and Techniques

In general [1] outlines abstraction & modelling principles and techniques applicable in any phase (domain, requirements and software design) of computing systems development: (a) property vs. model–oriented descriptions, (b) representation and operation abstraction, (c) denotations vs. computations (operational semantics), (d) hierarchies and compositions, (e) contexts, states and configurations, (f) applicative (functional), imperative, logic and parallel specification “programming”, and many other specification principles and techniques. In this short paper we, unfortunately, must omit relating each and every one of these principles and techniques to protocol engineering as for example practised in this paper.
4.2. Domain Modelling Principles and Techniques

[1, 2] outline a number of domain modelling principles and techniques: Modelling (i) static and dynamic, (ii) temporal and spatial, (iii) discrete, continuous and chaotic, and (iv) tangible and intangible attributes; modelling (v–vi) stake–holders and their perspectives; and modelling (vii) the intrinsics, (viii) the support technologies, (ix) the management & organisation, (x) the rules & regulations, and (xi) the human (correct, sloppy and criminal) behaviour facets.

Some relations were: (i) The The Citizen Smart Card PMR, PMJ, and Info information repositories were basically Static, whereas the Acc, Sec and Tick securities instrument (ie. money) repositories were Dynamic. (ii) There were Temporal relations between E–Business and The Citizen Smart Card transactions that determined crucial properties of protocols, and there we Spatial properties of the domain: Citizens gathering PMJ information “from far afield”, and traders trading across geographical distance — all necessitating distribution of information and processing. (iii) The examples mostly exemplified Discrete systems. The fact that the stock market may behave Chaotically, or that electro cardiograms portray Continuous phenomena is of no interest for the systems such as we choose to see them. (iv) Most PMR information is Tangible: “Feelings” of being in bad health may be indicated by informal utterings in a PMR, but are Intangible. Whether to order or not to order a product, and in case of ordering, then from whom, may be Intangible attributes, but, as for the others, are of little concern to our protocol designs.

(v) We took a rather limited view of stake–holders, focusing only on patients, bank account holders, securities buyers and sellers, and commuters, respectively only on traders (consumers, retailers, wholesaler and producers). We did not consider health–care professionals, staff of financial institutions, transport company staff (drivers, etc.), nor the specific clerks, etc. of traders. (vi) And we took in account only a rather limited perspective. Enough to focus on the essentials for this article: The protocol development.

(vii) The citizen behaviour of Section 2.1.2 as well as the trader behaviour of Section 2.2.8 can be said to reflect the very basics, the Intrinsics of citizens and traders vis–a–vis the issues dealt with. (viii) We did not illustrate domain Support Technologies—prior to the advent of The Citizen Smart Card and E–Business. Once new requirements are expressed wrt. existing The Citizen Smart Card or E–Business system, then their card and terminal, respectively their assumed Internet technologies
become support technologies. (ix) Past *Management & Organisation*, past “bureaucracies” tended to prevent such “free” mechanisms as The Citizen Smart Card and E–Business. (x) The *Rules & Regulations* of government supervised health-care, financial institutions and transport was not considered at all. Some such rules may affect protocols, however. (ix) We did consider *Human Behaviour* in both case examples — as they seem to influence, very strongly several aspects of protocol design: Loosing or stealing cards, issuing trading transactions “out of order” or forgetting to respond to such transactions. We refer to [1, 2].

4.3. Requirements Modelling
Principles and Techniques

We have mentioned some requirements construction principles and techniques: Projection, instantiation, extension, and initialisation. They were illustrated in Section 2. We refer, in general to [1, 4].

4.4. Software Design
Principles and Techniques

Domain and some, often all, interface requirements lead to software architecture design: A *software architecture*, to us, reflects exactly the issues with which the users are immediately and tangibly concerned. And these issues have to do with the domain, and with most of the interface. Machine requirements, and remaining interface requirements lead to *program organisation* design: They are intangible as far as the end–user is concerned, and may differ between computing and communication platforms. We refer to [1, 5].

5. CONCLUSION

This paper has — obviously — been written by a person who is not a researcher nor a practitioner of *protocol engineering*. Instead the author is a researcher and practitioner of the more “general” field of *software engineering* — if one can claim that such a broad field can be justified, let alone exists!

5.1. What Have We Achieved?

What we have tried to do is to cast new light on protocol engineering. We claim, and we have shown, we believe, that it is useful — to the extent of being necessary — to first study and document the domain from which applications (also of protocols) emerge. We have indicated
how protocols emerge from requirements, and how requirements emerge from domains.

5.2. What Remains to be Done?

The two examples were formulated as challenges. We have, ourselves, developed sketches and draft formalisations of some protocols, but would like to see others tackle the problem cases raised here!

Acknowledgements

The author gratefully acknowledges the kind invitation by Prof. Myungchul Kim (Inform. & Comm. Univ., Korea), Dr. Byungmool Chin (Electronics Telecomm. Res. Inst., Korea), and Dr. Sungown Kang (Korea Telecom, Korea). Thanks to them, and Prof. Sam Chanson (HKU-S&T), I was challenged into formulating my thoughts on protocols and protocol engineering as emanating from domain and requirements engineering.

References

[1] Dines Bjørner. Software Engineering: Theory & Practice. (Publisher is being contacted), 2002. This book represents the author’s Chef d’œvres — the summary of more than 25 years of research, development and teaching.


A PKI-BASED END-TO-END SECURE INFRASTRUCTURE FOR MOBILE E-COMMERCE*

Tin-Wo Cheung and Samuel T. Chanson

Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong
{cswo, chanson}@cs.ust.hk

Abstract

The popularity of handheld mobile devices, the move towards electronic commerce, and deployment of the public key infrastructure in many parts of the world have led to the development of electronic commerce on mobile devices. The main challenge is the limited computing capacity on these devices for PKI-based end-to-end secure transactions. This paper presents a new architecture and protocol for authentication and key exchange as well as the supporting infrastructure that is suitable for the mobile environment. The system requirements and our solutions in addressing these requirements in the restrictive environment are discussed. An evaluation of the system performance is also included. The system has been implemented and is supporting some real-life applications.

Keywords: wireless security, key exchange protocols, mobile e-commerce, PKI end-to-end security, smart card, GSM Short Message Service

* This work was supported by an ISF grant from the Hong Kong Innovation and Technology Commission (AF/153/99).
1. INTRODUCTION

1.1 Background

The rapid advances in wireless mobile communication technology and the growth of electronic commerce have naturally led to the development of electronic commercial services on the wireless medium through mobile phones. In Hong Kong as well as many parts of the world, the population of mobile phone users has more than doubled the number of Internet users. The popularity and high penetration rate of the mobile phone can be attributed to its convenience, ease of use, and low cost of ownership and operation. For business transactions conducted on electronic means, security is a major concern. Both the Internet and the wireless network are public networks and considered to be insecure, where messages can be eavesdropped, captured, modified and inserted by intruders. Intruders may also impersonate as legitimate parties for personal gain. Therefore, some mechanism is needed to guarantee the confidentiality, authenticity and integrity of the transmitted messages. Internationally, the Public Key Infrastructure (PKI) is accepted as an effective means to tackle the above security problem.

In January 2000, Hongkong Post [4] started its digital certification service and became the first recognized Certification Authority (CA) in Hong Kong. The Electronic Transactions Ordinance enacted in March the same year gives electronic signatures the same legal stature as handwritten signatures. Therefore, as in a number of other countries, a sound and legal public key infrastructure has been established in Hong Kong that is ready for commercial activities on the electronic medium.

1.2 Objective of Work

Our objective is to develop a PKI-based open infrastructure that supports end-to-end secure electronic transactions through mobile phones. We are not just looking for a theoretical solution; rather the solution must be practical, efficient, and implementable. Besides the security concerns, efficiency and availability of supporting hardware products are also important. The main challenge in this project is the limited capacity of the mobile device, especially the mobile phone [10, 12, 13, 14, 15, 17]. About the only computing facilities available on the mobile phone are those offered by the SIM card which has a very slow speed processor and little memory (of the order of 16 KB). The resources on current SIM cards are not sufficient to perform general PKI-based authentication. Moreover, the wireless network is error-prone and slow compared to wired networks. We have designed a set of key exchange and authentication protocols that can run on a thin client model.
The infrastructure consists of a number of servers offering supporting services for on-line purchase and payment in addition to authentication. Security is end-to-end rather than link-by-link.

The rest of the paper is organized as follows. Section 2 describes related works by others. The requirements of our system are given in Section 3 while Section 4 provides an overview of the system architecture. Section 5 presents our key exchange and authentication protocols, and Section 6 discusses some system implementation issues. System performance is presented in Section 7, and finally Section 8 concludes the paper.

2. RELATED WORKS

For secure digital communication, the communicating parties need to be sure of the identity of one another and the contents of their messages must not be tampered with in any way even if the communication network is insecure. An authentication mechanism is needed to satisfy the first requirement and a secret key exchange mechanism is needed to achieve the second goal. This is known as the authenticated key exchange (AK) problem, and a variation is called the authenticated key exchange with key confirmation (AKC) problem in [9]. Both problems are concerned with key agreement among the communicating parties. The AK protocol makes sure the key can be successfully exchanged only if the identities of the communication parties are genuine. However, it does not check if the exchanged key really works. The AKC protocol further adds messages to verify the key being established is correct. In this paper, we shall use the term AKC protocol to refer to the authentication and key exchange protocol in general.

2.1 Categories of AKC protocols

There is a vast amount of literature on AKC protocols. In a survey article [20], Clark et al. categorize AKC protocols according to criteria such as the cryptographic approach used, whether a trusted third party is present, and the number of protocol messages exchanged. In this section, we propose an alternative approach and categorize the published AKC protocols based on the methodologies used in authentication and key exchange. Most existing work deals with two communicating parties only.

2.1.1 Key exchange with Implicit Authentication

This class of protocols mainly apply the Diffie-Hellman [11] key exchange algorithm for exchanging keys between the communication parties. In this case, when two parties need to establish a session key, they exchange their
public keys. An identical data can be generated by each party by combining its own private key with the counterpart’s public key (see Formula (A)). Since only these two parties are able to generate the secret data, proof of the ability to generate the data implicitly authenticates the participants.

The Beller-Chang-Yacobi [12] protocol is designed for the mobile environment, in which the client is a low-power portable mobile device. Let $PK_j$ and $SK_j$ be the public key and the corresponding private key of the base station respectively.

![Figure 1. Beller-Chang-Yacobi Protocol](image1.png)

When both parties have successfully verified the certificates of their counterparts, they can calculate the value $\eta$ using the Diffie-Hellman technique:

$$\eta \equiv (PK_j)^{SK_i} \equiv (PK_i)^{SK_j} \pmod{N} \quad ---- (A)$$

A session key $sk$ is calculated by symmetrically encrypting a random number $x$ with key $\eta$, i.e., $sk = f(\eta, x)$.

The idea is similar to Diffie-Hellman in that a common session key can only be calculated by those with the corresponding private keys. The certificates ($Cert_i$ and $Cert_j$) in the protocol are for the purpose of ensuring the public keys used are valid.

Carlsen [13] and Zheng [14] have pointed out the weakness of this protocol due to the weak authenticity mechanism. The identities of both parties have not been authenticated even after message 2 has been received. This means either one can be an impersonator at this point without being detected.
2.1.2 Explicit Authentication by Challenge-Response with key exchange

The simplest way to authenticate a party is the “Challenge-Response” method. The responding party holds a secret, such as a secret symmetric key or a private key, that enables him to answer the challenge correctly. When some random data are embedded in the challenge and/or response messages, these data can be used to generate the session key after successful authentication.

The Aziz-Diffie protocol [15] is another certificate based AKC protocol. The difference between Beller-Chang-Yacobi and Aziz-Diffie is that the Aziz-Diffie protocol is based on the challenge-response mechanism. Therefore, we regard it as an authentication-oriented protocol with key-exchange functionality, while the Beller-Chang-Yacobi protocol is a key-exchange oriented protocol with implicit authentication functionality. The main design goal of Aziz-Diffie is to achieve both privacy of wireless data communication as well as mutual authentication between the communicating parties. The protocol can be summarized in the following diagram:

![Aziz-Diffie Protocol Diagram](image)

**Figure 2. Aziz-Diffie Protocol**

The mobile client initiates the protocol by sending $N_m$ in message 1 as the first challenge to the base station. The signature in message 2 contains the element $N_m$ which proves ownership of the private key of the base station $b$ whose certificate $Cert_b$ is also passed to Client $m$ in message 2. At the same time, the random number $X_b$ in message 2 is encrypted by the public key of $m$ as a challenge to the mobile client. In message 3, the mobile client responds to this challenge by decrypting it with its own private key and then encrypts it with the public key of the base station $b$. Since only the holder of the private key $m$ can decrypt $X_b$ correctly, the base station can authenticate the mobile
user after receiving message 3. As a result, mutual authentication is achieved. The session key is calculated by $X_b \text{ XOR } X_m$.

The Aziz-Diffie protocol is a typical challenge-response AKC protocol. However, many other proposals take a hybrid approach. In [10, 16], both Diffie-Hellman factors and challenge-response pairs are involved in the protocols messages. Therefore, mutual authentication as well as key establishment are achieved simultaneously.

2.1.3 Explicit Authentication by synchronised data with key exchange

Another approach to explicitly authenticate a party is by using some kind of synchronized data. The principle is basically the same as that of the challenge-response approach. The difference is instead of random numbers, synchronized data such as timestamps from synchronized clocks or counters are used. It is important to ensure the freshness of the synchronized data to avoid replay attacks. In [14], timestamps are used. However, in reality, it is difficult to guarantee the clocks are synchronized.

The Lin-Harn [17] protocol employs a chained hash technique for generating the synchronized data. Before a session key can be established, a registration process is required where the mobile client generates a group of hash chains, signs the “tail” of each chain and then sends to the base station for registration. The client can prove its identity by revealing a previous image of the registered hash chain. Due to the non-reversible property of the hash function, the ability to present a previous image in the chain implies ownership (generator) of the chain. The verifier can verify a received image by hashing it a number of times and compare the result to the ‘tail’ image.

In the Lin-Harn protocol, the synchronized data as well as the challenge is the last revealed image in the hash chain. The verifier accepts an image only if it is a previous image of the last revealed one. The ability to provide a valid response to this challenge implies the ability to reverse the hashing operation to recover the previous images. Since only the generator of the hash chain knows the “head” image of the chain, the authenticity requirement can be satisfied. Note that in this protocol a single message is enough for authentication.

2.2 Standardized AKC protocols

In order to make use of the AKC protocols in real life, standardization of some generally accepted protocols is essential. Normally, the standardized protocols are independent of the cryptographic algorithm. Negotiation of parameters is generally required at the beginning of the protocol.
2.2.1 Secure Socket Layer (SSL) [1]/Transport Layer Security (TLS) [7]

SSL/TLS is a well-known standard that is generally used for secure Internet web browsing. Its handshake protocol is designed to address the AKC problem between the SSL server and SSL client. The authentication mechanism is based on the challenge-response approach discussed above.

There are two important concepts of SSL: SSL session and SSL connection. A SSL session associates a client and a server. It is created through a handshake protocol involving negotiation of version and algorithms to be used in the session. After the handshake protocol, both the client and the server will be able to calculate a secret data for the SSL session called the master secret. Every SSL connection is a peer-to-peer logical connection built on top of a SSL session for data exchange in the application layer. In a SSL connection, the two peers share a set of secrets derived from the master secret of the SSL session. This set of secrets is used for encrypting the exchanged data and creating the message authentication codes (MACs) used to ensure data integrity.

The main purpose of the SSL connection is to enhance the efficiency of the protocol by reusing the master secret of a SSL session in different SSL connections between the same client and server. The handshake protocol for SSL connection establishment contains fewer messages and involves no public key cryptographic operations which are CPU intensive.

SSL is suitable for wired Internet environment. However, for restrictive devices such as the mobile phone or smart card, the protocol is too heavy for implementation. Moreover, a SSL session can only be reused for connections between the same pair of client-server. If a client tends to communicate with different servers, a new SSL session must be establishment for each server.

2.2.2 Wireless Transport Layer Security (WTLS) [8]

WTLS is the security layer protocol for the Wireless Application Protocol (WAP) architecture to provide confidentiality, data integrity and authenticity of the communicating applications. The design of WTLS bears a close resemblance to the design of TLS 1.0. Because of the low bandwidth and unreliability of the wireless network, WTLS is concerned with datagram support, optimized handshake and a key refreshment scheme.

One limitation of WTLS is it only provides point-to-point security in most cases. Figure 3 shows a general configuration of a WAP network. WTLS provides a secure channel between the WAP user and the WAP gateway. But it does not protect the communication all the way to the Wireless Markup Language (WML) server. Typically, a SSL session is established between the WAP gateway and the WML server if data security is a concern. In reality, WAP gateways are maintained by mobile operators and WML servers are
owned by service providers. Therefore, WTLS normally only provides a point-to-point secure solution to the service provider (WML sever) and the end user. If end-to-end security is required, either the WAP gateway has to be maintained by a trusted party or the service provider must maintain the WAP gateway herself.

2.3 Discussion

Different AKC protocols have been designed to tackle different issues, such as security of the protocol, restricted computational power of the client or privacy of the customers. However, few of the proposed solutions have considered utilizing the existing mobile system for implementing security adds-on, and as far as we know only the WTLS protocol has been implemented and deployed outside of the laboratory. The practical constraints and problems, therefore, have rarely been addressed in the literature. WTLS, however, does not provide end-to-end security between the user and the service provider.

In this work, the constraints of implementing an AKC protocol on the existing GSM mobile network using existing hardware products are discussed. A trusted third party is introduced in our AKC protocol to share the workload of certificate verification with the mobile client. Our design considerations are on security, restricted processing resources of the client, number of messages required and the size of each message packet.

3. SYSTEM REQUIREMENTS

In our system, users conduct mobile electronic transactions from a GSM dual-slot cellular phone. The phone has an additional smart card slot besides the GSM SIM card. The second smart card is chosen to be a Java Card [3] with 16K EEPROM and a cryptographic co-processor which can perform RSA cryptographic functions, triple-DES cipher, SHA-1 hashing and random
number generation. All AKC protocol processing as well as transaction operations are performed on the processor chip on-board the smart card. The GSM SIM card is only responsible for providing the wireless communication channel and invoking the applications on the second card. Message packets between the phone and the base station are transmitted through SMS (short message service) of GSM. A SMS packet has a size limitation of 140 bytes.

4. OVERVIEW OF SYSTEM ARCHITECTURE

Due to the scarce resource for both memory space and computational power, the mobile equipment (ME) is incapable of verifying a X.509 digital certificate to authenticate a service provider. We have developed a server called the User Authentication Server (UAS) to act as a trusted third party to assist the mobile client to authenticate and exchange keys with the service provider, which is named the PKI End-to-end Secure Module (PESM). The following diagram shows the system architecture. Each component will now be described below.

Figure 4. Overview of the system architecture

4.1 SMS Gateway and Mobile Electronic Service Server (MESS)

The SMS Gateway and MESS together act as an interface between the wireless and wired networks. MESS interprets the header of message packets and routes the packets to the proper MEs and servers. It is unable to read the message contents since they are encrypted at source.
4.2 Mobile client

The mobile client is a portable device which in our case is a dual slot GSM phone and a smart card with cryptographic functionality. Each user is required to have his own digital certificate issued by a valid Certification Authority (CA). The corresponding private key is stored in the user’s second slot smart card. Moreover, we require the UAS’s public key be pre-loaded on the card as well. In subsequent sections, the mobile client is abbreviated as ME (Mobile Equipment) for simplicity.

4.3 User Authentication Server (UAS)

The UAS is a centralized server that should be operated by a trusted third party. Its role is to help the ME to authenticate the party it is communicating with. First, mutual authentication is performed between the ME and UAS. Then, the UAS authenticates the PESM on behalf of the ME. Following that, a PESM session key is exchanged between the ME and PESM to establish an end-to-end secure communication channel.

4.4 PKI End-to-end Secure Module (PESM)

PESM is a server operated by the service provider. It is responsible for ensuring security at the application level, includes authentication, confidentiality, integrity and non-repudiation. For authentication, it performs the handshake protocol to authenticate the UAS or optionally authenticate the mobile client and establishes a session key. For confidentiality, it encrypts and decrypts messages sent and received from the mobile client using the established session key. Furthermore, it verifies the Message Authentication Code (MAC) of each message to guarantee integrity. For non-repudiation, it verifies the digital signature of a message if it is present.

4.5 Certificate repository

The certificate repository is a service provided by the CA which allows the public to access the issued digital certificates. It is usually implemented by a LDAP server on which object records can be searched by subjects. The UAS and PESMs will access this server from time to time to retrieve digital certificates for verification purposes.
5. PROTOCOLS

5.1 Design Criteria

Our AKC protocol is based on a 3-tiered model involving the ME, UAS and PESM. With the assistance of the UAS, the ME and PESM establish a session key between them for encryption and MAC calculation. Prior to key establishment, authentication is required between the UAS and ME, and then the UAS and PESM. For certain applications, the PESM may want to authenticate the client (ME) in order to determine the privileges allowed. If this is not necessary, the PESM normally does not need to authenticate the client. The ME and UAS, however, must authenticate the PESM to ensure they are talking to the right service provider.

The communication channel between the ME and MESS is furnished by the GSM SMS (Short Message Service) service. SMS is a datagram based packet switching channel in which message packets can be corrupted, delayed or lost. The maximum size of each message packet is 140 bytes and there is a significant delay associated with each SMS packet transmission. Furthermore, the slow processing speed of the ME and the limited memory space for program code and data on the ME’s second slot smart (of the order of 16K Bytes) should also been taken into account in the design of the protocol. Therefore, the protocol should satisfy the following requirements:

- The size of each message packet on the wireless network must be less than 140 bytes
- The number of messages to be transmitted on the wireless network should be minimized
- The process on the client side must be simplified in order to minimize the code size and processing time on the second slot smart card
- The protocol must be able to tolerate the errors on the wireless transmission channel

5.2 Description of Protocols

Our protocol is divided into 2 phrases, namely: UAS Session Establishment and PESM Session Establishment. This is illustrated in the following diagram (Figure 5).
We use the following notations in describing our protocol:

Table 1. Symbols used in protocol description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDₚ</td>
<td>A unique identifier of entity P</td>
</tr>
<tr>
<td>Eₚ[x]</td>
<td>Encrypt x by P’s public key</td>
</tr>
<tr>
<td>Certₚ</td>
<td>Digital certificate of P</td>
</tr>
<tr>
<td>ESKEY{x}</td>
<td>Encrypt by symmetric key block cipher in CBC mode (3DES) with the key “KEY”</td>
</tr>
<tr>
<td>hash{x}</td>
<td>Hash the value x</td>
</tr>
<tr>
<td>f(x)</td>
<td>Some kind of one-way function for session key diversification</td>
</tr>
<tr>
<td>N ∈ R {0, 1}^k</td>
<td>Randomly generate k bits of binary data N</td>
</tr>
<tr>
<td>A</td>
<td>B or A,B</td>
</tr>
</tbody>
</table>

5.2.1 UAS Session establishment

The session is established between the ME and UAS using a two-pass AKC protocol based on a general challenge-response authentication mechanism (see Figure 6).

Figure 6 UAS Session Establishment
Table 2. Definition of terms used in UAS Session Establishment protocol

<table>
<thead>
<tr>
<th>Items</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver</td>
<td>Version of the protocol</td>
</tr>
<tr>
<td>MachAttr</td>
<td>Configuration attribute of the ME (e.g., Language)</td>
</tr>
<tr>
<td>Na</td>
<td>A random number generated by ME</td>
</tr>
<tr>
<td>Seq</td>
<td>A random number generated by ME as the starting sequence number of this session</td>
</tr>
<tr>
<td>Nb</td>
<td>A random number generated by UAS</td>
</tr>
<tr>
<td>USKey</td>
<td>UAS Session Key calculated from $f(Na</td>
</tr>
<tr>
<td>KeyPolicy</td>
<td>A value defining the lifetime of USKey</td>
</tr>
</tbody>
</table>

The ME initiates the establishment of a secure session with the UAS by performing the following operations:

1. Randomly generates $Na$ and $Seq$.
2. Encrypts $(\text{hash}\{\text{Cert UAS}\}, \text{ID ME}, Na, Seq)$ using UAS’s public key.
3. Composes and sends $ukey\_session\_req$ to UAS.

When UAS receives the $ukey\_session\_req$ message, it should

1. Decrypt the message using its own private key.
2. Check if $\text{hash}\{\text{Cert UAS}\}$ is the fingerprint of its current certificate. If the check fails, the protocol cannot be continued since the ME does not have the correct public key of the UAS.
3. Randomly generate $Nb$ and calculate $USKey$ (UAS session key) from $f(Na||Nb)$.
4. Determine the lifetime of the session key and specify it in the value of $KeyPolicy$.
5. Compose and send $ukey\_session\_resp$ to ME.

On receiving the $ukey\_session\_resp$ message the ME verifies the validity of the message by generating its own value of $Hash\{\text{Ver, ID ME, USKey, KeyPolicy}\}$ and comparing it with the one in the message.

Since only the valid UAS can decrypt $ukey\_session\_req$ to get the value of $Na$, ME can authenticate UAS by checking the correctness of the $ukey\_session\_req$ message. If the message is correct, ME accepts $USKey$ and $KeyPolicy$.

In the above protocol, only one-way authentication of UAS is achieved. Adversaries can impersonate the ME by creating its own $ukey\_session\_req$ message. Therefore, the UAS does not accept this newly established session yet. Instead, it stores the state parameters (i.e., $USKey$, $KeyPolicy$ and $Seq$) of the session as a pending state and switch to the current state only after the ME has further authenticated itself in the $PESM Session Establishment$ protocol that follows.
5.2.2 PESM Session Establishment

After the UAS session has been established (either in pending state or current state), the ME may start the PESM Session Establishment protocol in order to establish a secure communication session with the service provider. A request is sent by the ME to the UAS specifying which PESM it would like to talk with. The UAS then communicates with the target PESM on behalf of the ME.

Before the UAS can start the key exchange protocol with a PESM, it may have to interact with the PESM to find out the key exchange mode required and exchange the related certificates. If the information is already known then this step can be skipped.

The PESM can choose from two authentication modes of session key establishment when it receives an enquiry: Server Authentication and Client Authentication. Server Authentication means the PESM does not need to authenticate the ME. Otherwise, Client Authentication mode is used.

### Server Authentication

If Server Authentication mode is selected, the protocol runs as follows:

![Protocol for PESM Session establishment in Server authentication mode](image)

Table 3. Definition of terms used in PESM Session Establishment protocol

<table>
<thead>
<tr>
<th>Items</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nx</td>
<td>A random number generated by ME</td>
</tr>
<tr>
<td>Seq</td>
<td>Sequence number of this session</td>
</tr>
<tr>
<td>NM1</td>
<td>A random number generated by ME</td>
</tr>
</tbody>
</table>
The ME initiates the protocol with the following actions:

1. Randomly generates \( NM1 \) and \( Nx \) and calculates \( UEKey \) from \( f(USKey \| Nx) \).
2. Increments \( Seq. \)
3. Encrypts \( (Seq, ID_{PESM}, NM1) \) using \( UEKey \).
4. Composes and sends \( pkey\_session\_req \) to UAS.

When UAS receives the \( pkey\_session\_req \) message, it

1. Computes \( UEKey \) using the received \( Nx \) and its own \( USKey \).
2. Decrypts the message using \( UEKey \).
3. Checks if the value of \( Seq \) is valid. UAS will only accept \( Seq \) if it is larger than the last accepted \( Seq \) but falls within a certain predefined range. This mechanism is to avoid intruder’s attack by replaying the \( pkey\_session\_req \) message.
4. If this message is valid, UAS switches session state from ‘pending’ to ‘current’.
5. If UAS has no information about the mode of authentication or the certificate of PESM, a \( pconnect\_query \) message is sent to the PESM. After receiving the \( pconnect\_ansA \) response, UAS checks if the PESM’s certificate was issued by one of the CAs listed in the non-empty \( CertReq \). If it does not check out, the session cannot be established.
6. Randomly generates \( NM2 \).
7. Encrypts the elements in the \( pconnect\_authA \) message using PESM’s public key and sends the message to PESM.

On receiving the \( pconnect\_authA \) message, the PESM

1. Decrypts the message using its private key.
2. Checks if the UAS certificate fingerprint in the message matches that of the certificate.
3. Randomly generates \( NM3 \) and computes the \( PSKey \) by \( f(NM1\|NM2\|NM3) \).
4. Determines the lifetime of the \( PSKey \) and assigns the value of \( KeyPolicy \).
5. Encrypts \( (Ver, SRN, KeyPolicy) \) using \( PSKey \).
6. Composes the \( pconnect\_finishA \) message, encrypts it using the public key of UAS and sends to UAS.
On receiving the message `pconnect_FinishA`, the UAS can authenticate the PESM by checking if the values of `NM1` and `NM2` are the same as what were sent in the `pconnect_authA` message. This is, again, a simple challenge-response mechanism since the values of `NM1` and `NM2` can only be obtained by the holder of PESM’s private key. After authenticating the identity of PSEM, the UAS forwards the needed1 data to ME needed to calculate the PESM session key.

Note that the message type of the response by PESM indicates the mode of authentication. A `pconnect_ansA` message means server authentication is required. This step may be skipped if UAS already has this information.

**Client Authentication**

If the PESM requires client authentication, a `pconnect_ansB` message is sent instead of `pconnect_ansA`. The only difference in message packets contents between these two modes is that the public key of ME is used to encrypt the message sent from PESM. As a consequence, only the ME is able to calculate `PSKey`. The protocol runs as follows:

![Protocol for PESM Session Establishment in Server authentication mode](image)

**5.3 Discussion**

All authentication mechanisms in the protocols are achieved by the “challenge-response” approach. In both the UAS and PESM Session protocols, two messages are exchanged on the wireless network, which is the minimal number for challenge-response. However, this can achieve one-way
authentication only. For mutual authentication, at least 3 messages are needed. Our approach of achieving mutual authentication with the minimum number of messages is to combine the last response with the message in the next protocol. In other words, completion of the authentication procedure is postponed until the next protocol. For example, in the UAS Session Establishment protocol, authentication of ME is not achieved until the start of the following PESM Session Establishment protocol. The same is done in the PESM Session Establishment protocol. Because of this, PESM and UAS should put the unauthenticated session in the pending state until confirmation can be made.

Another interesting property of this protocol is that since every protocol is a 2-pass protocol, the servers (i.e., UAS in the UAS Session Establishment protocol and PESM in the PESM Session Establishment protocol) are stateless. This is particularly important from the efficiency point of view since the servers do not need a waiting state.

6. IMPLEMENTATION ISSUES

This AKC protocol is designed to shift as much computational load as possible from the mobile device (ME) to the servers without compromising security. The infrastructure consists of a number of components as described in Section 4. In this section, some practical issues on the implementation of these components are discussed. There are 4 major entities involved in a transaction: mobile client, mobile operator, service provider and trusted third party.

As mentioned earlier, the mobile client is composed of three components: a dual slot GSM mobile phone, a GSM Subscriber Identity Module (SIM) which is a smart card with SIM Application Toolkit [21] enabled, and a second slot smart card. We have defined a set of protocols between the SIM and second slot cards so that they can communicate with each other. The workflow is controlled by the second slot card. The SIM and mobile phone together provide a user interface and communication channel for the applications on the second slot card. In our implementation, the second slot smart card is a Java Card [3] which is compliant with the Java Card 2.1 standards. The card has a crypto co-processor on-board that supports public key cryptographic functions in hardware. The Java card was chosen for its flexibility and ease of programming; and the crypto engine is needed for speed in executing the RSA operations. We have tried three different brands of this type of card to test the generality of the infrastructure. The code size of the protocol on the card is about 6~7 Kbytes.

For the mobile operator, a prototype of the MESS (see Figure 4) is implemented. This is a server that interfaces the wireless network with the wired network. It is physically connected to a SMS gateway and also the
Internet. A set of message headers is defined for the MESS to route the
associated messages from the SMS gateway to the appropriate destinations.

Because of the 140-byte limitation of the SMS message packet, all
messages in our protocol are designed to satisfy this constraint. However, for
the few cases when an application data message exceeds this limit, it will be
broken down into two or more SMS packets. MESS is responsible to combine
the fragmented SMS packets back to a single message according to the
information in the headers.

7. PERFORMANCE

We have measured the various segments of time needed to execute the
protocols on the mobile device in order to analyze the performance of the
protocols and determine the system’s bottleneck.

First we examine the time required for transmitting a SMS message (see
Table 4). This time mainly depends on the performance of the mobile handset
and the GSM SIM card. Different products can exhibit very different
performance. The purpose of the study is not to compare the service provided
by different mobile operators or to compare the performance of different
mobile phone and SIM card products. We are only interested in gaining some
ideas on the amount of time needed by each process involved in the
transmission of a short message.

Table 4. Timing for SMS transmission

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed overhead for sending each SMS packet which is independent of packet size (depends only on the handset)</td>
<td>4.43 ~ 6.39</td>
</tr>
<tr>
<td>Transmission delay of 140 bytes of data</td>
<td>1.75</td>
</tr>
<tr>
<td>Processing delay of 140 of bytes data (handset dependent)</td>
<td>0.22 ~ 0.28</td>
</tr>
<tr>
<td>Total amount of time for a handset to send a 140-byte SMS packet (1.75+0.22+4.43) ~ (1.75+0.28+6.39)</td>
<td>6.4 ~ 8.42</td>
</tr>
</tbody>
</table>

As can be seen, the bulk of the time is in transmitting the SMS packet onto
the wireless network from the handset no matter how small the size of the
packet. Therefore, it is obvious the priority is to minimize the number of SMS
messages used by the protocols rather than minimizing the packet size.

Another factor affecting the execution time of the protocol is the
processing delay of the second slot smart card. Since the processing speed of
the processor chip on the smart card chip is very slow compared with that of a
PC, it may induce a significant delay on protocol execution. Again the
processing time is highly dependent on the smart card product. We have
compared three brands of Java Card with cryptographic function (see Table
5).
Table 5. Timing for smart card operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation time of the second slot card by the handset</td>
<td>1.3</td>
</tr>
<tr>
<td>Brand of Java Card</td>
<td>A</td>
</tr>
<tr>
<td>RSA Private Key Decryption (Key length = 1024 bits)</td>
<td>0.58</td>
</tr>
<tr>
<td>RSA Public Key Encryption (Key length = 1024 bits)</td>
<td>0.19</td>
</tr>
<tr>
<td>SHA-1 hash on 64 bytes of data</td>
<td>0.16</td>
</tr>
<tr>
<td>Triple-DES encryption in CBC mode on 64 bytes of data</td>
<td>0.28</td>
</tr>
<tr>
<td>Triple-DES encryption in ECB mode on 64 bytes of data</td>
<td>0.21</td>
</tr>
<tr>
<td>Random number generation (64 bytes)</td>
<td>0.43</td>
</tr>
</tbody>
</table>

For the UAS Session Establishment protocol, the overall processing time spent by the mobile client is about 18–22s (two SMS messages and four second slot card invocations with the associated processing time on the smart card). On the other hand, the processing time incurred by the UAS, which is implemented on a Pentium III 550MHz PC, is only about 0.1s and is negligible compared to that of the mobile client. The total time to establish a UAS session is the sum of the processing times and the queuing delay at the SMS gateway which varies from time to time depending on the traffic intensity. Since the major part of the processing time is spent on SMS transmission and invocation of the second slot card, the delay time for the PESM Session Establishment protocol is roughly the same as that for the UAS Session Establishment protocol.

We can see that performing an AKC protocol on the GSM mobile wireless is unacceptable for time critical applications. If the mobile phone starts with the UAS Session Establishment protocol and then the PESM Session Establishment protocol and lastly the application data, it would likely take about 1 minute to finish the process. A way to reduce this time is to reuse the session keys which have been previously established. In our protocol, both the UAS session key and PESM session key are kept for a period of time determined by the KeyPolicy value assigned by the UAS and PESM. Normally, a session key can be used for hundreds of times before refreshment since the time needed to break a 128-bit session key using the traditional method is of the order of tens of years. In this case, the AKC protocols will not need to be executed most of the time before a transaction. The application data is directly encrypted by the established session key stored in memory on the second slot smart card and transmitted. The time needed for the transaction is about 9 seconds if no digital signature is required or 18 seconds otherwise.
8. CONCLUSIONS

In this paper AKC protocols are categorized according to their authentication and key exchange approaches. We then presented a new set of AKC protocols that are practical and optimized for the GSM wireless network with SMS as the carrier. The protocols involve three parties - the mobile client, the service provider, and a trusted third party- and provide PKI-based end-to-end security between the client and the service provider. It is one of the few systems that has been implemented and an analysis of its performance has been given.

To evaluate the applicability of the protocols to electronic commerce activities, we have implemented a payment application on the infrastructure. The payment system makes use of the secure features of the protocols to protect the credit card number transmitted from the second slot smart card on the mobile client to the payment server of the merchant’s acquirer bank. The application is being used in real-life for purchase and payment. The customers have found the system convenient to use and practical. The performance is acceptable and so far no security problem has been reported.

REFERENCES

A FAMILY OF RESOURCE-BOUND REAL-TIME PROCESS ALGEBRAS

Insup Lee
University of Pennsylvania, Philadelphia, PA (lee@cis.upenn.edu)

Jin-Young Choi
Korea University, Seoul, Korea (choi@formal.korea.ac.kr)

Hee Hwan Kwak
Synopsys Inc., Beaverton, OR (hkwak@synopsys.com)

Anna Philippou
University of Cyprus, Cyprus (annap@cs.ucy.ac.cy)

Oleg Sokolsky
University of Pennsylvania, Philadelphia, PA (sokolsky@saul.cis.upenn.edu)

Abstract This paper describes three real-time process algebras, ACSR, PACSR and ACSR-VP. ACSR is a resource-bound real-time process that supports synchronous timed actions and asynchronous instantaneous events as well as the notions of resource, priority, exception, and interrupt. PACSR is a probabilistic extension of ACSR with resources that can fail and associated failure probabilities. ACSR-VP extends ACSR with value passing between processes and parameterized process definitions. This paper also provides three simple real-time system examples to illustrate the expressive power and analysis technique of each process algebra.

Keywords: Real-time process algebra, probabilistic process algebra, value-passing process algebra, schedulability analysis, real-time systems, resource-bound process algebra.

1. INTRODUCTION

Reliability in real-time systems can be improved through the use of formal methods for the specification and analysis of timed behaviors. Recently, there has been a spate of progress in the development of real-time formal methods. Much of this work falls into the traditional categories of untimed systems such as temporal logics, assertional methods, net-
based models, automata theory and process algebras. In this paper, we provide an overview of the family of resource-bound real-time process algebras that we have developed.

Process algebras, such as CCS [12], CSP [7], Acceptance Trees [5] and ACP [2], have been developed to describe and analyze communicating, concurrently executing systems. They are based on the premises that the two most essential notions in understanding complex dynamic systems are concurrency and communication [12]. The most salient aspect of process algebras is that they support the modular specification and verification of a system. This is due to the algebraic laws that form a compositional proof system which enable the verification of a whole system by reasoning about its parts. Process algebras are being used widely in specifying and verifying concurrent systems.

Algebra of Communicating Shared Resource (ACSR) introduced by Lee et al. [10], is a timed process algebra which can be regarded as an extension of CCS. The timing behavior of a real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources. Most current real-time process algebras adequately capture delays due to process synchronization; however, they abstract out resource-specific details by assuming idealistic operating environments. On the other hand, scheduling and resource allocation algorithms used for real-time systems ignore the effect of process synchronization except for simple precedence relations between processes. ACSR algebra provides a formal framework that combines the areas of process algebra and real-time scheduling, and thus, can help us to reason about systems that are sensitive to deadlines, process interaction and resource availability.

ACSR supports the notions of resources, priorities, interrupt, timeout, and process structure. The notion of real-time in ACSR is quantitative and discrete, and is accommodated using the concept of timed actions. Executing a timed action requires access to a set of resources and takes one unit of time. Resources are serially reusable, and access to them is governed by priorities. Similar to CCS, the execution of an event is instantaneous and never consumes any resource. The notion of communication is modeled using events through the execution of complementary events, which are then converted into an internal event. As with timed actions, priorities are also used to arbitrate the choice of several events that are possible at the same time. Although the concurrency model of CCS-like process algebras is based on interleaving semantics, ACSR includes interleaving semantics for events as well as lock-step parallelism for timed actions.
The computation model of ACSR is based on the view that a real-time system consists of a set of communicating processes that use shared resources for execution and synchronize with one another. The use of shared resources is represented by timed actions and synchronization is supported by instantaneous events. The execution of a timed action is assumed to take one time unit and to consume a set of resources during the same time unit. Idling of a process is treated as a special timed action that consumes no resources. The execution of a timed action is subject to availability of the resources used in the timed action. The contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously. Unlike a timed action, the execution of an event is instantaneous and never consumes any resource. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are used to direct the choice when several events are possible at the same time.

We have extended ACSR into a family of process algebras, GCSR [1], Dense-time ACSR [4], ACSR-VP [9], and PACSR [15]. GCSR is a graphical version of ACSR which allows the visual representation of ACSR processes. Dense-time ACSR is an extension of ACSR with dense time. ACSR-VP extends ACSR with value-passing capability so that arbitrary scheduling problems can be specified and analyzed. Probabilistic ACSR allows the modeling of resource failure with probabilities.

The rest of the paper is organized as follows. Section 2 describes the basic computation model of ACSR and explains its notions of events and timed actions. Section 3 overviews the syntax and semantics of ACSR and describes a simple scheduling example. Section 4 explains PACSR and extends the same scheduling example with probabilistic resource failure. Section 5 describes ACSR-VP and shows how parametric scheduling analysis can be done using basically the same schedule example.

2. THE COMPUTATION MODEL

In our algebra there are two types of actions: those which consume time, and those which are instantaneous. The time-consuming actions represent one “tick” of a global clock. These actions may also represent the consumption of resources, e.g., CPUs, devices, memory, batteries in the system configuration. In contrast, the instantaneous actions provide a synchronization mechanism between a set of concurrent processes.

Timed Actions. We consider a system to be composed of a finite set of serially reusable resources, denoted by \( R \). An action that consumes
one “tick” of time is drawn from the domain $\mathbb{P}(\mathbb{R} \times \mathbb{N})$, with the restriction that each resource be represented at most once. As an example, the singleton action, $\{(r, p)\}$, denotes the use of some resource $r \in \mathbb{R}$ running at the priority level $p$. The action $\emptyset$ represents idling for one time unit, since no resuable resource is consumed.

We use $D_R$ to denote the domain of timed actions, and we let $A, B, C$ range over $D_R$. We define $\rho(A)$ to be the set of resources used by the action $A$; e.g., $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the use of resource $r$ in action $A$; e.g., $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$. By convention, if $r$ is not in $\rho(A)$, then $\pi_r(A) = 0$.

### Instantaneous Events

We call instantaneous actions events, which provide the basic synchronization in our process algebra. We assume a set of channels $L$. An event is denoted by a pair $(a, p)$, where $a$ is the label of the event, and $p$ is its priority. Labels are drawn from the set $\mathcal{L} \cup \hat{\mathcal{L}} \cup \{\tau\}$, where for all $a \in L$ $a? \in \mathcal{L}$ and $a! \in \hat{\mathcal{L}}$. We say that $a?$ and $a!$ are inverse labels. As in CCS, the special identity label, $\tau$, arises when two events with inverse labels are executed in parallel.

We use $D_E$ to denote the domain of events, and let $e, f$ and $g$ range over $D_E$. We use $l(e)$ and $\pi(e)$ to represent the label and priority, respectively, of the event $e$. The entire domain of actions is $D = D_R \cup D_E$, and we let $\alpha$ and $\beta$ range over $D$.

The executions of a process are defined by a timed labelled transition system (timed LTS). A timed LTS $M$ is defined as $\langle P, D, \to \rangle$, where (1) $P$ is a set of ACSR processes, ranged over by $P, Q$, (2) $D$ is a set of actions, and (3) $\to$ is a labeled transition relation such that $P \xrightarrow{\alpha} Q$ if the process $P$ may perform an instantaneous event or timed action $\alpha$ and then behave as $Q$.

For example, a process $P_1$ may have the following behavior: $P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} P_3 \xrightarrow{\alpha_3} \ldots$. That is, $P_1$ first executes $\alpha_1$ and evolves into $P_2$, which executes $\alpha_2$, etc. It takes no time to execute an instantaneous event. A timed action however is executed for exactly one unit of time.

### 3. ACSR

The following grammar describes the syntax of ACSR processes.

$$P ::= \ NIL \mid (a,n).P \mid A:P \mid P + P \mid P\|P \mid P \Delta^s_e (P, P, P) \mid P \backslash F \mid [P]I \mid P \| I \mid b \rightarrow P \mid C.$$  

The process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The process $(a, n)$. $P$ executes the instantaneous event $(a, n)$ and proceeds to $P$. The
process \(A:P\) executes a resource-consuming action during the first time unit and proceeds to \(P\). The process \(P + Q\) represents a nondeterministic choice between the two summands. The process \(P\|Q\) describes the concurrent composition of \(P\) and \(Q\): the component processes may proceed independently or interact with one another while executing events, and they synchronize on timed actions.

The scope construct, \(P \Delta^a_t (Q, R, S)\), binds the process \(P\) by a temporal scope and incorporates the notions of timeout and interrupts. We call \(t\) the time bound, where \(t \in \mathbb{N} \cup \{\infty\}\) and require that \(P\) may execute for a maximum of \(t\) time units. The scope may be exited in one of three ways: First, if \(P\) terminates successfully within \(t\) time-units by executing an event labeled \(a!\) where \(a \in L\), then control is delegated to \(Q\), the success-handler. Else, if \(P\) fails to terminate within time \(t\) then control proceeds to \(R\). Finally, throughout execution of this process construct, \(P\) may be interrupted by process \(S\).

In \(P \setminus F\), where \(F \subseteq L\), the scope of channels in \(F\) is restricted to process \(P\), and thus, components of \(P\) may use these labels to interact with one another but not with \(P\)'s environment. The construct \([P]_I\), \(I \subseteq R\), produces a process that reserves the use of resources in \(I\) for itself, extending every action \(A\) in \(P\) with resources in \(I - \rho(A)\) at priority 0. \(P \setminus I\) hides the identity of resources in \(I\) so that they are not visible on the interface with the environment. That is, the operator \(P \setminus I\) binds all free occurrences of the resources of \(I\) in \(P\). This binder gives rise to the sets of free and bound resources of a process \(P\). Process \(b \rightarrow P\) represents the conditional process: it performs as \(P\) if boolean expression \(b\) evaluates to true and as NIL otherwise. Process constant \(C\) with process definition \(C \overset{\text{def}}{=} P\) allows standard recursion.

The Structured Transition System. The informal account of behavior just given is made precise via a family of rules that define the labeled transition relations on processes. The semantics is defined in two steps. First, we develop the unconstrained transition system, where a transition is denoted as \(P \overset{\alpha}{\rightarrow} P'\). Within “\(\rightarrow\)” no priority arbitration is made between actions; rather, we subsequently refine “\(\rightarrow\)” to define our prioritized transition system, “\(\rightarrow \pi\).” The precise semantics rules are omitted but can be found in [3].

The prioritized transition system is based on preemption, which incorporates our treatment of synchronization, resource-sharing, and priority. The definition of preemption is straightforward. Let “\(\prec\)”, called the preemption relation, be a transitive, irreflexive, binary relation on actions. Then for two actions \(\alpha\) and \(\beta\), if \(\alpha \prec \beta\), we can say that “\(\alpha\) is preempted
by $\beta$.” This means that in any real-time system, if there is a choice between executing either $\alpha$ or $\beta$, $\beta$ will always be executed.

There are three cases to consider [3]: The first case is for the two timed actions, $\alpha$ and $\beta$, that compete for common resources. Here, the preempted action $\alpha$ may use a superset of $\beta$’s resources. However, $\beta$ uses all the resources at least at the same priority level as $\alpha$. Thus, for any resource $r$ in $\rho(\alpha) - \rho(\beta)$, the priority of $r$ in $\alpha$ must be zero in order that $\beta$ may preempt $\alpha$ since $\pi_r(B)$ is, by convention, 0 when $r$ is not in $B$. Also, $\beta$ uses at least one resource at a higher level. For instance, $\{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\}$ but $\{(r_1, 2), (r_2, 1)\} \not\prec \{(r_1, 7)\}$.

The second case is for the two events with the same label. Here, an event may be preempted by another event sharing the same label, but with a higher priority. For example, $(\tau, 1) \prec (\tau, 2)$, $(a, 2) \prec (a, 5)$, and $(a, 1) \not\prec (b, 2)$ if $a \neq b$.

The third case is when an event and a timed action are comparable under “$\prec$.” Here, if $n > 0$ in an event $(\tau, n)$, we let the event preempt any timed action. For instance, $\{(r_1, 2), (r_2, 5)\} \prec (\tau, 2)$, but $\{(r_1, 2), (r_2, 5)\} \not\prec (\tau, 0)$.

We define the prioritized transition system “$\rightarrow_{\pi}$,” which simply refines “$\rightarrow$” to account for preemption.

**Definition 1** The labeled transition system “$\rightarrow_{\pi}$” is defined as follows: $P \xrightarrow{\alpha}_{\pi} P'$ if and only if (1) $P \xrightarrow{\alpha} P'$ is an unprioritized transition, and (2) There is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$.

**Analysis of Real-Time Systems in ACSR.** Within the ACSR formalism we can conduct two types of analysis for real-time scheduling: validation and schedulability analysis. Validation shows that a given specification correctly models the required real-time scheduling discipline, such as Rate Monotonic and Earliest-Deadline-First. Schedulability analysis determines whether or not a real-time system with a particular scheduling discipline misses any of its deadlines. The validation and schedulability analysis of a real-time system can be carried out using the equivalence of ACSR processes.

Equivalence between ACSR processes is based on the concept of bisimulation [14] which compares the computation trees of two processes. Using the theory found in [12], it is straightforward to show that there exists a largest such bisimulation over “$\rightarrow_{\pi}$,” which we denote as “$\sim_{\pi}$.” This relation is an equivalence relation, and is a congruence with respect to ACSR’s operators [3].
When comparing processes, we often find that because different objectives were pursued in formulating the two process expressions (perhaps simplicity of expressions for one, and efficiency for the other), the internal synchronization actions of the two processes are not identical. Consequently, even though the two processes may display identical “external” behavior (i.e., non-$\tau$ event labels and timed action steps), there may be $\tau$ actions in one process that do not correspond directly with $\tau$ actions in the other. (Recall that synchronization replaces the complementary event labels with a single $\tau$ event.) For those situations where matching of external behaviors is sufficient a weaker form of equivalence, weak bisimulation [12], is used. It is straightforward to prove the existence of a largest weak bisimulation $\approx_\pi$ over “$\rightarrow_\pi$” in a manner analogous to the case for bisimulation. Weak bisimulation, $\approx_\pi$, is an equivalence relation (though not a congruence) for ACSR that compares observable behaviors of processes.

**Example.** Throughout the paper, we will use a simple example from the area of schedulability analysis. The example describes a set of periodic tasks scheduled according to the Rate Monotonic (RM) scheduling policy. This policy assigns static priorities to the tasks in the inverse proportion to their periods. As a syntactic convenience, we allow ACSR processes to be parameterized by a set of index variables. Each index variable is given a fixed range of values. This restricted notion of parameterization allows us to represent collections of similar processes concisely. For example, the parameterized process

$$P_t = t < 2 \rightarrow (a_t, t).P_{t+1}, t \in \{0..2\}$$

is equivalent to the following three processes:

$$P_0 = (a_0, 0).P_1, \quad P_1 = (a_1, 1).P_2, \quad P_2 = \text{NIL}.$$  

As shown in Section 5, the addition of parameterized process definition in ACSR-VP allows us to get rid of this kind of parameterization.

The example is constructed as follows. We have two tasks, Task$_1$ and Task$_2$. Task$_i$ has period $p_i$ and execution time $e_i$. The deadline for each task is equal to its period. Both tasks share the same processor, modeled by the resource cpu. No other tasks use the processor.

Each Task$_i$ idles until it is awakened by the operating system by the start$_i$ event and starts competing for the processor. At each time unit, the task may either get access to the processor or, if it is preempted by a higher-priority task, it idles until the next time unit. Once the necessary amount (i.e., $e_i$) of execution time is accumulated, the task returns to the initial state and waits for the next period.
In order to detect missed deadlines, we also model the task dispatcher, which initiates the tasks according to their periods. For each Task \( i \), there is a process Dispatch \(_i\), which sends the start\(_i\) event to the respective task every \( p_i \) time units. If the task cannot accept the event - that is, if it has not completed its execution - the dispatcher deadlocks.

The complete specification is shown below. In the specification of a task, \( i \) is the task number and \( j \) is the accumulated execution time. We assume that the tasks have distinct periods and are ordered by decreasing periods, so we can use the task number as the priority for processor access.

\[
\begin{align*}
\text{System} & \triangleq [(\text{Dispatch}_1|\text{Dispatch}_2|\text{Task}_1 | \text{Task}_2)\setminus\{\text{start}_1, \text{start}_2\}]_{\{\text{cpu}\}} \\
\text{Dispatch}_i & \triangleq (\text{start}_i!, i).D_{i,0} \quad i = \{1, 2\} \\
D_{i,k} & \triangleq \begin{cases} 
 k < p_i & \rightarrow \{\} : D_{i,k+1} \\
 k = p_i & \rightarrow \text{Dispatch}_i 
\end{cases} \quad i = \{1, 2\}, k = \{0, p_i\} \\
\text{Task}_i & \triangleq (\text{start}_i?, 0).P_{i,0} + \{\} : \text{Task}_i \quad i = \{1, 2\} \\
P_{i,j} & \triangleq \begin{cases} 
 j < e_i & \rightarrow \{(\} : P_{i,j} \\
 +\{(\text{cpu}, i)\} : P_{i,j+1} 
\end{cases} \quad i = \{1, 2\}, j = \{0, e_i\}
\end{align*}
\]

ACSR analysis techniques allow us to verify the schedulability of a system of tasks for fixed values of parameters \( e_i \) and \( p_i \). The correctness criterion being that a resulting process does not deadlock can be checked either by deciding the behavioral equivalence of the process to the process that idles forever, or by performing reachability analysis on the state space of the process to search for deadlock states. For example, we considered two sets of tasks. The task set with parameters \( e_1 = 2, p_1 = 5, e_2 = 1, p_2 = 2 \) does not exhibit any deadlock, while the set \( e_1 = 2, p_1 = 3, e_2 = 1, p_2 = 2 \) has a deadlock and thus is not schedulable.

4. PROBABILISTIC ACSR

PACSR (Probabilistic ACSR) extends the process algebra ACSR by associating with each resource a probability. This probability captures the rate at which the resource may fail. Since instantaneous events in PACSR are identical to those of ACSR, we only discuss timed actions, which now can account for resource failure.

**Timed Actions.** As in ACSR, we assume that a system contains a finite set of serially reusable resources drawn from the set \( R \). We also consider set \( \bar{R} \) that contains, for each \( r \in R \), an element \( \bar{r} \), representing
the failed resource \( r \). We write \( \mathcal{R} \) for \( \mathcal{R} \cup \overline{\mathcal{R}} \). Actions are constructed as in ACSR, but now can contain both normal and failed resources. So now the action \( \{(r, p)\}, r \in \mathcal{R} \), cannot happen if \( r \) has failed. On the other hand, action \( \{\overline{(r, q)}\} \) takes place with priority \( q \) given that resource \( r \) has failed. This construct is useful for specifying recovery from failures.

**Resource Probabilities.** In PACSR we associate each resource with a probability at which the resource may fail. In particular, for all \( r \in \mathcal{R} \) we denote by \( p(r) \in [0, 1] \) the probability of resource \( r \) being up, while \( p(\overline{r}) = 1 - p(r) \) denotes the probability of \( r \) failing. Thus, the behavior of a resource-consuming process has certain probabilistic aspects to it which are reflected in the operational semantics of PACSR. For example, consider the process \( \{(cpu, 1)\} : NIL \), where resource \( cpu \) has probability of failure \( 1/3 \), i.e., \( p(cpu) = 1/3 \). Then, with probability \( 2/3 \), resource \( cpu \) is available and thus the process may consume it and become inactive, while with probability \( 1/3 \) the resource fails and the process deadlocks.

**Probabilistic Processes.** The syntax of PACSR processes is the same as that of ACSR. The only extension concerns the appearance of failed resources in timed actions. Thus, it is possible on one hand to assign failure probabilities to resources of existing ACSR specifications and perform probabilistic analysis on them, and, on the other hand, to ignore failure probabilities and apply non-probabilistic analysis of PACSR specifications.

As with ACSR, the semantics of PACSR processes is given in two steps. At the first level, a transition system captures the nondeterministic and probabilistic behavior of processes, ignoring the presence of priorities. Subsequently, this is refined via a second transition system which takes action priorities into account.

The unprioritized semantics is based on the notion of a world, which keeps information about the state of the resources of a process. Given a set of resources \( Z \subset \mathcal{R} \), the set of possible worlds involving \( Z \) is given by \( \mathcal{W}(Z) = \{Z' \subseteq \mathcal{Z} \cup \overline{\mathcal{Z}} \mid x \in Z' \text{ iff } \overline{x} \not\in Z'\} \), that is, it contains all possible combinations of the resources in \( Z \) being up or down. Given a world \( W \in \mathcal{W}(Z) \), we can calculate the probability of \( W \) by multiplying the probabilities of every resource in \( W \).

Behavior of a given process \( P \) can be given only with respect to the world \( P \) is in. A configuration is a pair of the form \( (P, W) \in \text{Proc} \times 2^\mathcal{R} \), representing a PACSR process \( P \) in world \( W \). The semantics is given in terms of a labeled transition system whose states are configurations and
whose transitions are either probabilistic or nondeterministic. We write \( S \) for the set of configurations.

The intuition for the semantics is as follows: for a PACSR process \( P \), we begin with the configuration \((P, \emptyset)\). As computation proceeds, probabilistic transitions are performed to determine the status of resources which are immediately relevant for execution but for which there is no knowledge in the configuration’s world. Once the status of a resource is determined by some probabilistic transition, it cannot change until the next timed action occurs. Once a timed action occurs, the state of resources has to be determined anew, since in each time unit resources can fail independently from any previous failures. Nondeterministic transitions (which can be events or actions) may be performed from configurations that contain all necessary knowledge regarding the state of resources.

We partition the set \( S \) into probabilistic configurations \( S_p \) and nondeterministic configurations \( S_n \). A configuration \((P, W)\) is included in \( S_n \) if every resource that can be used in a first step of \( P \) is included in \( W \). Transitions for a configuration \((P, W) \in S_n \) are determined in the same way as in ACSR for \( P \), except that a transition labeled by an action \( A \) can be taken if every resource \( r \in R \) that appears in \( A \) is also contained in \( W \). The probabilistic transition relation takes probabilistic configurations into non-deterministic configurations.

We illustrate the rules of the semantics with the following example. Consider the process \( P = \{(r_1, 1), (r_2, 1)\} : P_1 + (e?, 1).P_2 \). The immediately relevant resources of \( P \) are \( \{r_1, r_2\} \). From the probabilistic configuration \((P, \{r_1\})\), where we know that \( r_1 \) is up, but have no information about \( r_2 \), we have two probabilistic transitions that determine the state of \( r_2 \): \((P, \{r_1\}) \xrightarrow{p(r_2)} (P, \{r_1, r_2\}) \) and \((P, \{r_1\}) \xrightarrow{p(f_2)} (P, \{r_1, f_2\}) \). Both of these configurations are nondeterministic since we have full information about the relevant resources. Further, \((P, \{r_1, r_2\})\) has two nondeterministic transitions: \((P, \{r_1, r_2\}) \xrightarrow{(r_1, 1)} (P_1, \emptyset) \) and \((P, \{r_1, r_2\}) \xrightarrow{(e?, 1)} (P_2, \{r_1, r_2\}) \). The other configuration allows only one transition: \((P, \{r_1, f_2\}) \xrightarrow{(e?, 1)} (P_2, \{r_1, f_2\}) \), since \( r_2 \) is failed. Note that a probabilistic transition always leads to a nondeterministic configuration. A nondeterministic transition may lead to either nondeterministic configuration or a probabilistic one.

**Probabilistic Analysis Techniques.** We have defined a probabilistic weak bisimulation [16], which allows us to compare observable behaviors of PACSR processes similar to the case of ACSR. In addition, probabilistic information embedded in the probabilistic transitions al-
lows us to perform quantitative analysis of PACSR specifications. In particular, we can compute the probability of reaching a given state or a deadlocked state.

**Example.** We illustrate the utility of PACSR in the analysis of fault-tolerance properties by slightly extending the example of Section 3. We consider the same set of tasks running on a processor with an intermittent fault. At any time unit, the processor may be running, in which case the higher-priority task executes normally, or it may be down, in which case none of the tasks execute. We modify the specification of a task to add the alternative behavior where a task can perform an action that contains the failed cpu resource and does not increase its execution time.

We apply the probabilistic analysis to the task set we considered in Section 3: $e_1 = 2$, $p_1 = 5$, $e_2 = 1$, $p_2 = 2$. Even though the task set is schedulable under perfect conditions, in the presence of failures the tasks may still miss their deadlines. Given the probability of a processor failure, we can compute the probability that a deadline is missed. The following list of pairs show results of the experiments we ran. The first element of each pair is the cpu failure probability and the second is the probability of a missed deadline: 

\{(0,0), (0.005, 0.025), (0.01, 0.050), (0.02, 0.100), (0.05, 0.250), (0.075, 0.367), (0.1, 0.473), (0.15, 0.650), (0.2, 0.780), (0.3, 0.926)\}.

5. **ACSR-VP**

ACSR-VP (ACSR with Value Passing) extends the process algebra ACSR described in Section 3 by allowing values to be communicated along communication channels. In this section we present ACSR-VP concentrating on its value-passing capabilities.

We assume a set of variables $X$ ranged over by $x, y$ and a set of values $V$ ranged over by $v$. Moreover, we assume a set $\text{Expr}$ of expressions
(which includes arithmetic expressions) and we let \( BExpr \subseteq Expr \) be the subset containing boolean expressions. We let \( e \) and \( b \) range over \( Expr \) and \( BExpr \), respectively, and we write \( z \) for a tuple \( z_1, \ldots, z_n \) of syntactic entities.

As in ACSR, ACSR-VP also has two types of actions: instantaneous events and timed actions. The notion of timed action is identical to that of ACSR. However, instantaneous events are extended to provide value passing in addition to synchronization. An event is denoted as a pair \((i, e_p)\) representing execution of action \( i \) at priority \( e_p \), where \( i \) ranges over the internal \( \tau \), the input event \( c?x \), and the output event \( c!e \).

The syntax of ACSR-VP processes is similar to that of ACSR except for \( C(\vec{x}) \).

\[
P ::= \text{NIL} \mid (a,n).P \mid A : P \mid P + P \mid P||P \mid P \triangleleft_t (P,P,P) \mid P:F \mid [P]_r \mid P\setminus I \mid b \rightarrow P \mid C(\vec{x}).
\]

In the input-prefixed process \((c?x,e).P\) the occurrences of variable \( x \) is bound. We write \( \text{fv}(P) \) for the set of free variables of \( P \). Each process constant \( C \) has an associated definition \( C(\vec{x}) \overset{\text{def}}{=} P \) where \( \text{fv}(P) \subseteq \vec{x} \) and \( \vec{x} \) are pairwise distinct. We note that in an input prefix \((c?x,e).P\), \( e \) should not contain the bound variable \( x \), although \( x \) may occur in \( P \).

An informal explanation of ACSR-VP constructs is similar to that of ACSR. The semantics of ACSR-VP process is also defined as a labeled transition system, similarly to that of ACSR. It additionally makes use of the following ideas: Process \((c!e_1,e_2).P\) transmits the value obtained by evaluating expression \( e_1 \) along channel \( c \), with priority the value of expression \( e_2 \), and then behaves like \( P \). Process \((c?x,p).P\) receives a value \( v \) from communication channel \( c \) and then behaves like \( P[v/x] \), that is, \( P \) with \( v \) substituted for variable \( x \). In the concurrent composition \((c?x,p_1).P_1||(c!v,p_2).P_2\), the two components of the parallel composition may synchronize with each other on channel \( c \) resulting in the transmission of value \( v \) and producing an event \((\tau,p_1+p_2)\).

**Symbolic Transition System.** Consider the simple ACSR-VP process \( P \overset{\text{def}}{=} (in?x,1).(out!x,1).\text{NIL} \) that receives a value along channel \( in \) and then outputs it on channel \( out \), and where \( x \) ranges over integers. According to traditional methods for providing semantic models for concurrent processes using transition graphs, process \( P \) is infinite branching, as it can engage in the transition \((in?n,1)\) for every integer value \( n \). Thus, standard techniques for analysis and verification of finite state systems cannot be applied to such processes. Several approaches, such as symbolic transition graphs and transition graphs with assignment, have been proposed to deal with this problem for various subclasses of
value-passing processes [6, 11, 13, 8]. We now briefly explain how to represent symbolic graphs with assignment for ACSR-VP processes. We only give an overview of the model and we refer to [8] for a complete discussion.

An SGA (Symbolic Graph with Assignment) is a rooted directed graph where each node $n$ has an associated finite set of free variables $fv(n)$ and each edge is labeled by a guarded action with assignment [11, 17]. Note that a node in SGA is an ACSR-VP term.

The notion of a substitution, which we also call assignment, is defined as follows. A substitution is any function $\theta: X \rightarrow \text{Expr}$, such that $\theta(x) \neq x$ for a finite number of $x \in X$. Given a substitution $\theta$, the support (or domain) of $\theta$ is the set of variables $D(\theta) = \{x|\theta(x) \neq x\}$. A substitution whose support is empty is called the identity substitution, and is denoted by Id. When $|D(\theta)| = 1$, we use $[\theta(x)/x]$ for the substitution $\theta$.

An SGA for ACSR-VP is a rooted directed graph where each node $n$ has an associated ACSR-VP term and each edge is labeled by a boolean predicate, an action, and an assignment, $(b, \alpha, \theta)$. Here we illustrate how to construct an SGA by an example. A set of rules for generating an SGA from an ACSR-VP term can be found in [8]. We use a transition $P \xrightarrow{b, \alpha, \theta} P'$ to denote that given the truth of boolean expression $b$, $P$ can evolve to $P'$ by performing actions $\alpha$ and putting into effect the assignment $\theta$. Consider the following process.

$$
P(x) \overset{\text{def}}{=} (a?y, 1).P'(x + 1, y)
$$

$$
P'(x, y) \overset{\text{def}}{=} (y \leq 2) \rightarrow (a!(x + y), 2).\text{NIL}
$$

The SGA for this process is shown below. Note how the value $x + 1$ is assigned along the edge from $P$ to $P'$.

An informal interpretation of the above SGA is to view each process node as a procedure with its respective formal parameters. An edge coming into a node corresponds to a call to the procedure with the actual parameters supplied by the assignment labeling the edge and input events. If some of the variables are missing from the assignment, they are taken to be the same as the variable of the same name in the source node of the edge. After the process has been “called” in this way, it evaluates the guards on the outgoing transitions, applies the preemption relation to the enabled transitions and selects one of the remaining transitions non-deterministically for the next step.
Symbolic Weak Bisimulation. The bisimulation relation for symbolic transition graphs is defined in terms of relations parametrized on boolean expressions, of the form $\varphi^b$, where $p \equiv^b q$ if and only if, for each interpretation satisfying boolean $b$, $p$ and $q$ are bisimilar in the traditional notion [8].

Let us compare the process $P$ described above with the following process $R$:

\[
P(x) \overset{\text{def}}{=} (a?y, 1).P'(x + 1, y)
\]
\[
P'(x, y) \overset{\text{def}}{=} (y \leq 2) \rightarrow (a!(x + y), 2).NIL
\]
\[
R(x') \overset{\text{def}}{=} (a?y', 1).R'(x', y')
\]
\[
R'(x', y') \overset{\text{def}}{=} (y' \leq 2) \rightarrow (a!(x' + y' + 1), 2).NIL
\]

The prioritized SGA for $R$ is a minor variation of the SGA for $P$. Applying the symbolic bisimulation algorithm for processes $P$ and $R$, we obtain the following predicate equation system.

\[
X_{00}(x, x') = \forall z \forall z' X_{11}(z, z', x + 1, x')
\]
\[
X_{11}(z, z', x, x') = z \leq 2 \rightarrow z' \leq 2 \land x + z = x' + z' + 1
\]
\[
\land z' \leq 2 \rightarrow z \leq 2 \land x' + z' + 1 = x + z
\]

This equation system can easily be reduced to the equation $X_{00}(x, x') \equiv x = x' + 1$, which allows us to conclude that $P(x)$ and $R(x')$ are bisimilar if and only if $x = x' + 1$ holds. In general, if we restrict to the domain of linear expressions, predicate equations obtained from the bisimulation algorithm can be solved using constraint logic programming and integer programming techniques [18].

**Example.** We revisit the example of Section 3 in order to conduct a more sophisticated, compared to ACSR, analysis allowed by ACSR-VP. With ACSR, the execution time and period of each task had to be fixed in order for the analysis to be performed. The symbolic semantics of ACSR-VP allows us to perform parametric analysis of processes. When we treat parameters of tasks as free variables of the specification, the scheduling problem can be restated in ACSR-VP as follows:

\[
\text{System}(e_1, e_2, p_1, p_2) \overset{\text{def}}{=} [(Dispatch_1(p_1) \mid Dispatch_2(p_2)) \mid Task_1(e_1)]
\]
\[
\mid Task_2(e_2)) \{\text{start}_1, \text{start}_2\}_{\text{cpu}}
\]
\[
\text{Dispatch}_i(p) \overset{\text{def}}{=} (\text{start}_i, t).D_i(0, p)
\]
\[
i \in \{1, 2\}
\]
\[
D_i(k, p) \overset{\text{def}}{=} k < p \rightarrow \emptyset : D_i(k + 1, p)
\]
\[
+ k = p \rightarrow \text{Dispatch}_i(p)
\]
\[
i \in \{1, 2\}
\]
\[
\text{Task}_i(e) \overset{\text{def}}{=} (\text{start}_i, \emptyset).P_i(0, e) + \{e : \text{Task}_i(e)\}
\]
\[
i \in \{1, 2\}
\]
\[
P_i(j, e) \overset{\text{def}}{=} j < e \rightarrow \{e : P_i(j, e)
\[
+ \{\text{cpu}, i\} : P_i(j + 1, e)\}
\]
\[
+ j = e \rightarrow \text{Task}_i(e)
\]
\[
i \in \{1, 2\}
\]
We can now perform parametric analysis of System for some or all of its parameters. Here we consider the process System\((2,1, p_1, p_2)\) Omitting the SGA for the example, we show the predicate equation system:

\[
\begin{align*}
X_1(p_1, p_2) &= X_2(0, 0, p_1, p_2) \\
X_2(j_{21}, j_{22}, p_1, p_2) &= X_3(0, 0, j_{21}, j_{22}, p_1, p_2) \\
X_3(j_{11}, j_{12}, j_{21}, j_{22}, p_1, p_2) &= \\
&= (j_{12} < p_1) \land (j_{21} < 1) \land (j_{22} < p_2) \land X_3(j_{11}, j_{12} + 1, j_{21} + 1, j_{22} + 1, p_1, p_2) \\
&+ (j_{12} < p_1) \land (j_{11} < 2) \land (j_{22} < p_2) \land X_3(j_{11} + 1, j_{12} + 1, j_{21}, j_{22} + 1, p_1, p_2) \\
&+ (j_{12} < p_1) \land (j_{11} = 2) \land (j_{22} < p_2) \land (j_{21} = 1) \land X_3(j_{11}, j_{12} + 1, j_{21}, j_{22} + 1, p_1, p_2) \\
&+ (j_{12} = p_1) \land (j_{11} = 2) \land X_3(0, 0, j_{21}, j_{22}, p_1, p_2) \\
&+ (j_{22} = p_2) \land (j_{21} = 1) \land X_3(j_{11}, j_{12}, 0, 0, p_1, p_2)
\end{align*}
\]

We have the additional condition \(p_2 < p_1\), which comes from the assumption made in Section 3 that processes are sorted by decreasing execution time. When we solve the system of equations under this condition, the set of values for the parameters is given by the pairs of integers \((p_1, p_2)\) satisfying \(p_1 > 3, p_2 > 1, p_1 > p_2\).

6. SUMMARY AND CURRENT WORK

We have presented three resource-bound real-time process algebras: ACSR, PACSR and ACSR-VP. ACSR employees a synchronous semantics for resource-consuming actions that take time and an asynchronous semantics for events that are instantaneous. ACSR was developed to handle schedulability analysis in a process-algebraic setting. PACSR supports the notion of probabilistic resource failures, whereas ACSR-VP extends ACSR with value-passing capability during communication and parameterized process definition. To illustrate their features, we have described and analyzed simple real-time systems using ACSR, PACSR, and ACSR-VP.

As mentioned in the introduction section, there are two more formalisms in the family of resource-bound real-time process algebras. One formalism is for visual specification of ACSR and the other is ACSR with dense time. We are currently developing a resource-aware process algebra to capture the notion of power consumption and resource constraints of embedded systems.

Acknowledgments. The authors gratefully acknowledge that the work described in this paper was influenced by the former members of Real-Time Systems Group at the University of Pennsylvania, including Hanène Ben-Abdallah, Patrice Brémond-Grégoire, Duncan Clarke, Rich Gerber, and Hong-Liang Xie. This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA ITO MO-
REFERENCES


Abstract
Survivability is the ability of a system to continue operating despite the presence of abnormal events such as accidental failures and malicious intrusions. Ensuring system survivability has increased in importance as critical infrastructures have become heavily dependent on computers. Examples of these infrastructures are utility, transportation, communication, and financial networks. Complicating the analysis of these networked systems is their inter-dependencies: a failure in one may trigger a failure in another.

In this talk I present a two-phase method for performing survivability analysis of networked systems. First, we inject failure and intrusion events into a system model, use model checking to verify it for fault- and service-related properties, and visually display the model’s effects with respect to a given property as a scenario graph. Then, we annotate the graphs with symbolic or numeric probabilities to enable reliability analysis using standard Markov Decision Process policy iteration algorithms. We use similar modeling and analysis techniques to do latency and cost-benefit analyses of these networked systems. We model dependencies among events using Bayesian Networks. We applied our two-phase method to two large cases studies from the financial industry and are currently applying it to a case study on intrusion detection systems.

This work is jointly done with Somesh Jha of the University of Wisconsin and Oleg Sheyner of Carnegie Mellon University. A full paper describing this work is in the Proceedings of the International Conference on Software Engineering, May 2001.

Keywords: survivability, fault-tolerance, security, reliability, model checking, Markov Decision Processes, Bayesian Networks, scenario graphs.