Model-driven High-level Programming of Embedded Systems: Selected papers from SLA++P’07 and SLA++P’08

Guest Editors: Florence Maraninchi, Michael Mendler, Marc Pouzet, Alain Girault, and Eric Rutten
Model-driven High-level Programming of Embedded Systems: Selected papers from SLA++P’07 and SLA++P’08
Model-driven High-level Programming of Embedded Systems: Selected papers from SLA++P’07 and SLA++P’08

Guest Editors: Florence Maraninchi, Michael Mendler, Marc Pouzet, Alain Girault, and Eric Rutten
Editor-in-Chief
Zoran Salcic, University of Auckland, New Zealand

Associate Editors

Sandro Bartolini, Italy
Neil Bergmann, Australia
Shuvra Bhattacharyya, USA
Ed Brinksma, The Netherlands
Paul Caspi, France
Liang-Gee Chen, Taiwan
Dietmar Dietrich, Austria
Stephen A. Edwards, USA
Alain Girault, France
Rajesh K. Gupta, USA
Susumu Horiguchi, Japan

Thomas Kaiser, Germany
Bart Kienhuis, The Netherlands
Chong-Min Kyung, Korea
Miriam Leeser, USA
John McAllister, UK
Koji Nakano, Japan
Antonio Nunez, Spain
Sri Parameswaran, Australia
Zebo Peng, Sweden
Marco Platzner, Germany
Marc Pouzet, France

S. Ramesh, India
Partha S. Roop, New Zealand
Markus Rupp, Austria
Asim Smailagic, USA
Leonel Sousa, Portugal
Jarmo Henrik Takala, Finland
Jean-Pierre Talpin, France
Jürgen Teich, Germany
Dongsheng Wang, China
Contents

Selected Papers from SLA++P 07 and 08 Model-Driven High-Level Programming of Embedded Systems, Florence Maraninchi, Michael Mendler, Marc Pouzet, Alain Girault, and Eric Rutten
Volume 2008, Article ID 376920, 2 pages

Lutin: A Language for Specifying and Executing Reactive Scenarios, Pascal Raymond, Yvan Roux, and Erwan Jahier
Volume 2008, Article ID 753821, 11 pages

Compilation and Worst-Case Reaction Time Analysis for Multithreaded Esterel Processing, Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden
Volume 2008, Article ID 594129, 21 pages

Formal Analysis Tools for the Synchronous Aspect Language Larissa, David Stauch
Volume 2008, Article ID 310147, 18 pages

Embedded Systems Programming: Accessing Databases from Esterel, David White and Gerald Lttgen
Volume 2008, Article ID 961036, 17 pages

SoC Design Approach Using Convertibility Verification, Roopak Sinha, Partha S. Roop, and Samik Basu
Volume 2008, Article ID 296206, 19 pages
Editorial

Selected Papers from SLA++P 07 and 08 Model-Driven High-Level Programming of Embedded Systems

Florence Maraninchi,1 Michael Mendler,2 Marc Pouzet,3 Alain Girault,4 and Eric Rutten4

1 VERIMAG Laboratory, 38610 Gieres, France
2 University of Bamberg, 96045 Bamberg, Germany
3 Laboratoire de Recherche en Informatique (LRI), Université Paris-Sud 11, 91405 Orsay Cedex, France
4 INRIA Grenoble - Rhône-Alpes, 38334 Saint Ismier Cedex, France

Correspondence should be addressed to Florence Maraninchi, florence.maraninchi@imag.fr

Received 31 December 2008; Accepted 31 December 2008

Model-based high-level programming of embedded systems has become a reality in the automotive and avionics industries. These industries place high demands on the efficiency and maintainability of the design process as well as on the performance and functional correctness of embedded components. These goals are hard to reconcile in the face of the increasing complexity of embedded applications and target architectures. Research efforts towards meeting these goals have brought about a variety of high-level engineering design languages, tools, and methodologies. Their strength resides in clean behavioral models with strong semantical foundations providing a rigorous way to go from a high-level description to mathematically certifiable executable code.

The most successful representatives of this trend of putting logic and mathematics behind design automation in embedded systems are synchronous languages; they have been receiving increasing attention in industry ever since they emerged in the 80s. Lustre, Esterel, and Signal are now widely and successfully used to program real-time and safety critical applications, from nuclear power plant management layer to Airbus air flight control systems. Their recent successes in the automatic control industry highlight the benefits of formal verification and automatic code generation from high-level models.

Model-based programming is making its way in other fields of software engineering too. Strong interest is emerging in component programming for large-scale embedded systems, in the link between simulation tools and compiler tools, in languages for describing the system and its environment, integrated tools for both compilation and simulation of more general models of communication and coordination, and so forth. The impact of such unifying methodologies will depend on the extent to which it will be possible to maintain the high degree of predictability and verifiability of system behavior that is the strength of the classic synchronous world.

List of Published Papers

This special issue features five very interesting papers. The first paper, “Lutin: a language for specifying and executing reactive scenarios,” is by P. Raymond et al. It introduces the Lutin language, which targets the description and the execution of constrained random scenarios for reactive systems. It does so by allowing the user to express, in a Lustre-like dataflow style, constraints on input/output relations. The language constructs are inspired by regular expressions and process algebra.

The second paper, “Compilation and worst-case reaction time analysis for multithreaded Esterel processing,” is by R. Von Hanxleden et al. It presents the compiling method used for Esterel programs onto the Kiel Esterel Processor (KEP), a multithreaded reactive architecture equipped with a dedicated instruction set to handle the Esterel features. On top of providing very efficient code, it is predictable, which allows the computation of the Worst Case Reaction Time (WCRT) of Esterel programs, an essential feature for real-time systems.
The third paper, “Formal analysis tools for the synchronous aspect language Larissa,” is by D. Stauch. It presents two tools for the formal analysis of the aspect language Larissa, which extends the Argos synchronous language. The first tool allows the combination of design-by-contract with Larissa aspects. The second tool allows to weave aspects in a less conflict-prone manner, therefore allowing the static detection of remaining conflicts statically.

The fourth paper, “Embedded systems programming: accessing databases from Esterel,” is by G. Luettgen and D. White. It presents two Application Programming Interfaces (APIs) which enable the use of relational databases inside Esterel programs. The first API is dedicated to database requests that can be answered very fast, and hence that complies to the synchrony hypothesis, while the second API is dedicated to database requests that must be handled asynchronously thanks to the external task mechanism of Esterel.

The fifth and final paper, “SoC Design Approach using Convertibility Verification,” is by R. Sinha et al. It addresses the compositional design of systems on chip from verified components, and particularly the issue of protocol converters enabling the matching of different components. Convertibility is verified using Kripke structures, model checking of ACTL temporal logic, and a tableau-base converter generation algorithm (Bamberg, Grenoble, and Paris, April 10th, 2009.).

Florence Maraninchi
Michael Mendler
Marc Pouzet
Alain Girault
Eric Rutten
Research Article

Lutin: A Language for Specifying and Executing Reactive Scenarios

Pascal Raymond, Yvan Roux, and Erwan Jahier

VERIMAG (CNRS, UJF, INPG), 2 avenue de Vignate, Gières 38610, France

Correspondence should be addressed to Pascal Raymond, pascal.raymond@imag.fr

Received 13 September 2007; Accepted 10 January 2008

Recommended by Michael Mendler

This paper presents the language Lutin and its operational semantics. This language specifically targets the domain of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. More precisely, it is dedicated to the description and the execution of constrained random scenarios. Its first use is for test sequence specification and generation. It can also be useful for early simulation of huge systems, where Lutin programs can be used to describe and simulate modules that are not yet fully developed. Basic statements are input/output relations expressing constraints on a single reaction. Those constraints are then combined to describe non deterministic sequences of reactions. The language constructs are inspired by regular expressions and process algebra (sequence, choice, loop, concurrency). Moreover, the set of statements can be enriched with user-defined operators. A notion of stochastic directives is also provided in order to finely influence the selection of a particular class of scenarios.

Copyright © 2008 Pascal Raymond et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

The targeted domain is the one of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. Examples of such systems are control/command in industrial process, embedded computing systems in transportation.

Testing reactive software raises specific problems. First of all, a single execution may require thousands of atomic reactions, and thus as many input vector values. It is almost impossible to write input test sequences by hand; they must be automatically generated according to some concise description. More specifically, the relevance of input values may depend on the behavior of the program itself; the program influences the environment which in turn influences the program. As a matter of fact, the environment behaves itself as a reactive system, whose environment is the program under test. This feedback aspect makes offline test generation impossible; testing a reactive system requires to run it in a simulated environment.

All these remarks have led to the idea of defining a language for describing random reactive systems. Since testing is the main goal, the programming style should be close to the intuitive notion of test scenarios, which means that the language is mainly control-flow oriented.

The language can also be useful for early prototyping and simulation, where constrained random programs can implement missing modules.

1.1. Our proposal: Lutin

For programming random systems, one solution is to use a classical (deterministic) language together with a random procedure. In some sense, nondeterminism is achieved by relaxing deterministic behaviors. We have adopted an opposite solution, where nondeterminism is achieved by constraining chaotic behaviors; in other terms, the proposed language is mainly relational not functional.

In the language Lutin, nonpredictable atomic reactions are expressed as input/output relations. Those atomic reactions are combined using statements like sequence, loop, choice or parallel composition. Since simulation (execution) is the goal, the language also provides stochastic constructs to express that some scenarios are more interesting/realistic than others.

Since the first version [1], the language has evolved with the aim of being a user-friendly, powerful programming language. The basic statements (inspired by regular expressions) have been augmented with more sophisticated control structures (parallel composition, exceptions) and
a functional abstraction has been introduced in order to provide modularity and reusability.

1.2. Related works

This work is related to synchronous programming languages [2, 3]. Some constructs of the language (traps and parallel composition) are directly inspired by the imperative synchronous language Esterel [4], while the relational part (constraints) is inspired by declarative languages like Lustre [5] and Signal [6].

Related works are abundant in the domain of models for nondeterministic (or stochastic) concurrent systems: Input/Output automata [7], and their stochastic extension [8] (stochastic extension of process algebra [9, 10]). There are also relations with concurrent constraint programming [11], in particular, with works that adopt a synchronous approach of time and concurrency [12, 13]. However, the goals are rather different; our goal is to maintain an infinite interaction between constraints generators, while concurrent constraint programming aims at obtaining the solution of a complex problem in a (hopefully) finite number of interactions.

Moreover, a general characteristic of these models is that they are defined to perform analysis of stochastic dynamic systems (e.g., model checking, probabilistic analysis). On the contrary, Lutin is specifically designed for simulation rather than general analysis. On one hand, the language allows to concisely describe, and then execute a large class of scenarios. On the other hand, it is in general impossible to decide if a particular behavior can be generated and even less with which probability.

1.3. Plan

The article starts with an informal presentation of the language. Then, the operational semantics is formally defined in terms of constraints generator. Some important aspects, in particular constraints solving, are parameters of this formal semantics; they can be adapted to favor the efficiency or the expressive power. These aspects are presented in the implementation section. Finally, we conclude by giving some possible extensions of this work.

2. OVERVIEW OF THE LANGUAGE

2.1. Reactive, synchronous systems

The language is devoted to the description of nondeterministic reactive systems. Those systems have a cyclic behavior; they react to input values by producing output values and updating their internal state. We adopt the synchronous approach, which here simply means that the execution is viewed as a sequence of pairs "input values/output values."

Such a system is declared with its input and output variables; they are called the support variables of the system.

Example 1. We illustrate the language with a simple "tracker" program that receives a boolean input (c) and a real input (t) and produces a real output (x). The high-level specification of the tracker is that the output x should get closer to the input t when the command c is true or should tend to zero otherwise. The header of the program is

\[
\text{system tracker (c: \text{bool}; t: \text{real}) returns (x: \text{real})} = \text{statement.}\]

The core of the program consists of a statement describing the program behavior. The definition of statement is developed later.

During the execution, inputs are provided by the system environment; they are called uncontrollable variables. The program reacts by producing outputs; they are called controllable variables.

2.2. Variables, reactions, and traces

The core of the system is a statement describing a sequence of atomic reactions.

In Lutin, a reaction is not deterministic; it does not define uniquely the output values, but states some constraints on these values. For instance, the constraint \((x > 0.0)\) and \((x < 10.0))\) states that the current output should be some value comprised between 0 and 10.

Constraints may involve inputs, for instance, \((x > t - 2.0)\) and \((x < t))\). In this case, during the execution, the actual value of t is substituted, and the resulting constraint is solved.

In order to express temporal constraints, previous values can be used; \(\text{pre id}\) denotes the value of the variable id at the previous reaction. For instance, \((x > \text{pre x})\) states that x must increase in the current reaction. Like inputs, \(\text{pre}\) variables are uncontrollable; during the execution, their values are inherited from the past and cannot be changed—this is the nonbacktracking principle.

Performing a reaction consists in producing, if it exists, a particular solution of the constraint. Such a solution may not exist.

Example 2. Consider the constraint

\[
(c \text{ and } (x > 0.0) \text{ and } (x < \text{pre x} + 10.0)),
\]

where c (input) and \(\text{pre x}\) (past value) are uncontrollable. During the execution, it may appear that c is false and/or that \(\text{pre x}\) is less than -10.0. In those cases, the constraint is unsatisfiable; we say that the constraint deadlocks.

Local variables may be useful auxiliaries for expressing complex constraints. They can be declared within a program:

\[
\text{local ident : type in statement.}
\]

A local variable behaves as a hidden output; it is controllable and must be produced as long as the execution remains in its scope.

2.3. Composing reactions

A constraint (Boolean expression) represents an atomic reaction; it defines relations between the current values of
the variables. Scenarios are built by combining such atomic reactions with temporal statements. We introduce the type trace for typing expressions made of temporal statements. A single constraint obviously denotes a trace of length 1; in other terms, expressions of type bool are implicitly cast to type trace when combined with temporal operators.

The basic trace statements are inspired by regular expression, and have following signatures:

(i) \texttt{fby: } trace \times trace \rightarrow trace \text{[sequence]}
(ii) \texttt{loop: } trace \rightarrow trace \text{[unbounded loop]}
(iii) | : trace \times trace \rightarrow trace \text{[nondeterministic choice]}

Using regular expressions makes the notion of sequence quite different from the one of Esterel, which is certainly the reference in control-flow oriented synchronous language [4]. In Esterel, the sequence (semicolon) is instantaneous, while the Lutin construct \texttt{fby} ”takes” one instant of time, just like in classical regular expressions.

Example 3. With those operators, we can propose a first version of our example. In this version, the output tends to resides in the initial value, and also in the fact that the system with those operators, we can propose a first command.

\[
\begin{align*}
((-100.0 < x) \land (x < 100.0)) \, \texttt{fby—initial constraint} \\
\text{loop}
\end{align*}
\]

\[
\begin{align*}
(c \land (x = 0.9*(\text{pre } x) + 0.1*t)) & \rightarrow x \text{ gets closer to } t \\
((x = 0.9*(\text{pre } x)) & \rightarrow x \text{ gets closer to } 0
\end{align*}
\]

Initially, the value of \(x\) is (randomly) chosen between \(-100\) and \(+100\), then forever, it may tend to \(t\) or to \(0\).

Note that, inside the loop, the first constraint (\(x\) tends to \(t\)) is not satisfiable unless \(c\) is true, while the second is always satisfiable. If \(c\) is false, the first constraint deadlocks. In this case, the second branch (\(x\) gets closer to \(0\)) is necessarily taken. If \(c\) is true, both branches are feasible: one is randomly selected and the corresponding constraint is solved.

This illustrates an important principle of the language, the reactivity principle, which states that a program may only deadlock, if all its possible behaviors deadlock.

2.4. Traces, termination, and deadlocks

Because of nondeterminism, a behavior has in general several possible first reactions (constraints). According to the reactivity principle, it deadlocks only if all those constraints are not satisfiable. If at least one reaction is satisfiable, it must “do something” we say that it is startable.

Termination, startability, and deadlocks are important concepts of the language; here is a more precise definition of the basic statements according to these concepts.

(i) A constraint \(c\), if it is satisfiable, generates a particular solution and terminates, otherwise it deadlocks.

(ii) \texttt{st1 \ fby \ st2} executes \texttt{st1}, and if and when it terminates, it executes \texttt{st2}. If \texttt{st1} deadlocks, the whole statement deadlocks.

(iii) \texttt{Loop st}, if \texttt{st} is startable, behaves as \texttt{st \ fby \ loop st}, otherwise it terminates. Indeed, once started, \texttt{st \ fby \ loop st} may deadlock if the first \texttt{st}, and so on. Intuitively, the meaning is “loop as long as starting a step is possible.”

(iv) \{\texttt{st1}\} \cdots \{\texttt{stn}\} randomly chooses one of the \texttt{startable} statements from \texttt{st1,..,stn}. If none of them are startable, the whole statement deadlocks.

(v) The priority choice \{\texttt{st1} \> \cdots \> \texttt{stn}\} behaves as \texttt{st1} if \texttt{st1} is startable, otherwise, behaves as \texttt{st2} if \texttt{st2} is startable and so on. If none of them are startable, the whole statement deadlocks.

(vi) Try \texttt{st1} do \texttt{st2} catches any deadlock occurring \textit{during the execution of} \texttt{st1} (not only at the first step). In case of deadlock, the control passes to \texttt{st2}.

2.5. Well-founded loops

Let us denote by \(\varepsilon\) the identity element for \texttt{fby} (i.e., the unique behavior such that \texttt{fby} \(\varepsilon = \varepsilon \texttt{fby} b = b\)). Although this “empty” behavior is not provided by the language, it is helpful for illustrating a problem raised by nested loops.

As a matter of fact, the simplest way to define the loop is to state that “\texttt{loop c}” is equivalent to “\texttt{c fby loop c}”, that is, try in priority to perform one iteration and if it fails, stop. According to this definition, nested loops may generate infinite and instantaneous loops, as shown in the following example.

Example 4.

\[
\text{loop(}\text{loop } c). \quad (5)
\]

Performing an iteration of the outer loop consists in executing the inner \texttt{loop(}\texttt{loop } c\texttt{)}. If \(c\) is not currently satisfiable, \texttt{loop } c\texttt{ terminates immediately and thus, the iteration is actually “empty”—it generates no reaction. However, since it is not a deadlock, this strange behavior is considered by the outer loop as a normal iteration. As a consequence, another iteration is performed, which is also empty, and so on. The outer loop keeps the control forever but does nothing.

One solution is to reject such programs. Statically checking whether a program will infinitely loop or not is undecidable, it may depend on arbitrarily complex conditions. Some over-approximation is necessary, which will (hopefully) reject all the incorrect programs, but also lots of correct ones. For instance, a program as simple as \texttt{“loop {\{loop a\} \ fby \ {loop b\}}”} will certainly be rejected as potentially incorrect.

We think that such a solution is too much restrictive and tedious for the user and we prefer to slightly modify the semantics of the loop. The solution retained is to introduce the well-founded loop principle; a loop statement may stop or continue, but if it continues it must do something. In other terms, empty iterations are dynamically forbidden.
The simplest way to explain this principle is to introduce an auxiliary operator \( st \), If \( st \) terminates immediately, \( st \) behaves as \( st \). The correct definition of \( loop \) \( st \) follows:

(i) if \( st \) is startable, it behaves as \( st \) 1 by \( loop \) \( st \),
(ii) otherwise \( loop \) \( st \) terminates.

### 2.6. Influencing non-determinism

When executing a nondeterministic statement, the problem of which choice should be preferred arises. The solution retained is that, if \( k \) out of the \( n \) choices are startable, each of them is chosen with a probability \( 1/k \).

In order to influence this choice, the language provides the concept of relative weights:

\[
\{st1 \text{ weight } w1 | \cdots | stn \text{ weight } wn\}. \tag{6}
\]

Weights are basically integer constants and their interpretation is straightforward. A branch with a weight 2 has twice the chance to be tried than a branch with weight 1. More generally, a weight can depend on the environment and on the past; it is given as an integer expression depending on uncontrollable variables. In this case, weight expressions are evaluated at runtime before performing the choice.

**Example 5.** In a first version (Example 3), our example system may ignore the command \( c \) with a probability 1/2. This case can be made less probable by using weights (when omitted, a weight is implicitly 1):

\[
\text{\( loop\) \{} \\
\quad (c \text{ and } (x = 0.9*(\text{pre } x) + 0.1*t)) \text{ weight } 9 \\
\quad (x = 0.9*(\text{pre } x)) \\
\}\]

In this new version, a true occurrence of \( c \) is missed with the probability 1/10.

Note that, weights are not only directives. Even with a big weight, a nonstartable branch has a null probability to be chosen, which is the case in the example when \( c \) is false.

### 2.7. Random loops

We want to define some loop structure, where the number of iterations is not fully determined by deadlocks. Such a construct can be based on weighted choices, since a loop is nothing but a binary choice between stopping and continuing. However, it seems more natural to define it in terms of expected number of iterations. Two loop “profiles” are provided as follows.

(i) \( \text{\( loop\)}[\text{min, max}]: \) the number of iterations should be between the constants \( \text{\( min \)} \) and \( \text{\( max \)} \).

(ii) \( \text{\( loop\)} \text{ \( av : sd \)}: \) the average number of iteration should be \( \text{\( av \)} \), with a standard deviation \( \text{\( sd \)} \).

Note that random loops, just like other nondeterministic choices, follow the *reactivity principle*; depending on deadlocks, looping may sometimes be required or impossible. As a consequence, during an execution, the actual number of iterations may significantly differ from the “expected” one (see Sections 4 and 5.3).

Moreover, just like the basic loop, they follow the well-founded loop principle, which means that, even if the core contains nested loops, it is impossible to perform “empty” iterations.

### 2.8. Parallel composition

The parallel composition of Lutin is synchronous: each branch produces, at the same time, its local constraints. The global reaction must satisfy the conjunction of all those local constraints. This approach is similar to the one of temporal concurrent constraint programming [12].

A parallel composition may deadlock for the following two reasons.

(i) Obviously, if one or more branches deadlock, the whole statement aborts.

(ii) It may also appear that each individual statement has one or more possible behaviours, but that none of the conjunctions are satisfiable, in which case the whole statement aborts.

If no deadlock occurs, the concurrent execution terminates, if and when all the branches have terminated (just like in the Esterel Language).

One can perform a parallel composition of several statements as follows:

\[
\{st1 \& \cdots \& \> stn \}. \tag{8}
\]

The concrete syntax suggests a noncommutative operator; this choice is explained in the next section.

### 2.9. Parallel composition versus stochastic directives

It is impossible to define a parallel composition which is fair according to the stochastic directives (weights), as illustrated in the following example.

**Example 6.** Consider the statement

\[
\{X \text{ weight } 1000|Y\} \& \> \{A \text{ weight } 1000|B\}, \tag{9}
\]

where \( x, a, x \land b, a \land y \) are all startable, but not \( x \land a \).

The higher priority can be given to

(i) \( x \land b \), but it would not respect the stochastic directive of the second branch;

(ii) \( a \land y \), but it would not respect the stochastic directive of the first branch;

In order to deal with this issue, the stochastic directives are not treated in parallel, but in *sequence*, from left to right.

(i) The first branch “plays” first, according to its local stochastic directives.

(ii) The next ones make their choice according to what has been chosen for the previous ones.

In the example, the priority is then given to \( x \land b \).
The concrete syntax (& >) has been chosen to reflect the fact that the operation is not commutative. The treatment is parallel for the constraints (conjunction), but sequential for stochastic directives (weights).

2.10. Exceptions

User-defined exceptions are mainly means for by-passing the normal control flow. They are inspired by exceptions in classical languages (Ocaml, Java, Ada) and also by the trap signals of Esterel.

Exceptions can be globally declared outside a system (exception ident) or locally within a statement, in which case the standard binding rules hold

\[
\text{exception ident in st.} \quad (10)
\]

An existing exception ident can be raised with the statement:

\[
\text{raise ident} \quad (11)
\]

and caught with the statement:

\[
\text{catch ident in st1 do st2.} \quad (12)
\]

If the exception is raised in st1, the control immediately passes to st2. The do part may be omitted, in which case the control passes in sequence.

2.11. Modularity

An important point is that the notion of system is not a sufficient modular abstraction. In some sense, systems are similar to main programs in classical languages. They are entry point for the execution but are not suitable for defining “pieces” of behaviors.

Data combinators

A good modular abstraction would be one that allows to enrich the set of combinators. Allowing the definition of data combinators is achieved by providing a functional-like level in the language. For instance, one can program the useful “within an interval” constraint;

\[
\text{let within}(x, \text{min}, \text{max} : \text{real}) : \text{bool} = (x >= \text{min}) \text{ and } (x <= \text{max}). \quad (13)
\]

Once defined, this combinator can be instantiated, for instance,

\[
\text{within}(a, 0.8, 0.9) \quad (14)
\]

or

\[
\text{within}(a + b, c - 1.0, c + 1.0). \quad (15)
\]

Note that, such a combinator is definitively not a function in the sense of computer science—it actually computes nothing. It is rather a well-typed macro defining how to build a Boolean expression with three real expressions.

Reference arguments

Some combinators specifically require support variables as argument (input, output, local). This is the case for the operator pre, and as a consequence, for any combinator using a pre. This situation is very similar to the distinction between “by reference” and “by value” parameters in imperative languages. Therefore, we solve the problem in a similar manner by adding the flag ref to the type of such parameters.

Example 7. The following combinator defines the generic first-order filter constraint. The parameter y must be a real support variable (real ref) since its previous value is required. The other parameters can be any expressions of type real.

\[
\text{Let } \text{f of } (y : \text{real}; \text{gain}, x : \text{real}) : \text{bool} = (y = \text{gain} \ast (\text{pre} y) + (1.0 - \text{gain} \ast x)). \quad (16)
\]

Trace combinators

User-defined temporal combinators are simply macros of type trace.

Example 8. The following combinator is a binary parallel composition, where the termination is enforced when the second argument terminates.

\[
\text{Let as long as, } (X, Y : \text{trace}) : \text{trace} =
\]

\[
\text{exception Stop in}
\]

\[
\text{catch Stop in}
\]

\[
X & > \{\text{Y fby raise Stop}\}
\]

Local combinators

A macro can be declared within a statement, in which case the usual binding rules hold; in particular, a combinator may have no parameter at all;

\[
\text{Let id ([params]) : type = statement in statement.} \quad (18)
\]

Example 9. We can now write more elaborated scenarios for the system of Example 3. For the very first reaction (line 2), the output is randomly chosen between −100 and +100, then the system enters its standard behavior (lines 3 to 14). A local variable a is declared, which will be used to store the current gain (line 3). An intermediate behavior (lines 4 to 6) is declared, which defines how the gain evolves; it is randomly chosen between 0.8 and 0.9, then it remains constant during 30 to 40 steps, and so on. Note that, this combinator has no parameter since it directly refers to the variable a. Lines 7 to 14 define the actual behavior; the user-defined combinator as long as runs in parallel the behavior gen_gain (line 8) with the normal behavior (9 to 11). In the normal behavior, the system works almost properly for about 1000 reactions; if c is true, x tends to t 9 times out of 10 (line 10), otherwise it tends to 0 (line 11). As soon as the normal behavior terminates, the whole parallel composition
(1) system tracker(c : bool; t : real) returns (x : real) =
(2) within(x, -100.0, 100.0) fby
(3) local a : real in
(4) let gen_gain(): trace = loop{
(5) within(a, 0.8, 0.9) fby loop[30, 40](a = pre a)
(6) } in
(7) as_long_as(
(8) gen_gain(),
(9) loop 1000: 100{
(10) (c and fof(x, a,t)) weight 9
(11) } fof(x, a, 0.0)
(12) }
(13) } fby
(14) loop fof(x, 0.7, 0.0)

Figure 1: A full example: the ”tracker” program.

Figure 2: An execution of the tracker program.

terminates (definition of as long as). Then, the system breaks down and x quickly tends to 0 (line 14).

Figure 2 shows the timing diagram, a particular execution of this program. Input values are provided by the environment (i.e., us) according to the following specification, the input t remains constant (150) and the command c toggles each about 100 steps.

3. SYNTAX SUMMARY

Figure 3 summarizes the concrete syntax of Lutin. The detailed syntax for expression is omitted. They are made of classical algebraic expressions with numerical and logical operators, plus the special operator pre. The supported type identifiers are currently bool, int, and real.

We do not present the details of the type checking, which is classical and straightforward. The only original check concerns the arguments of the loop profiles and of the weight directive, that must be uncontrollable expressions (not depending on output or local variables).

4. OPERATIONAL SEMANTICS

4.1. Abstract syntax

We consider here a type checked Lutin program. For the sake of simplicity, the semantics is given on the flat language. User-defined macros are inlined, and local variables are made global through some correct renaming of identifiers. As a consequence, an abstract system is simply a collection of variables (inputs, outputs, and locals) and a single abstract statement.

We use the following abstract syntax for statements, where the intuitive meaning of each construct is given between parenthesis:

\[
t ::= c \text{ (constraint), } e \text{ (empty behavior).}
\]
\[
| t \triangleleft e \text{ (empty filter).}\ t \cdot t \text{ (sequence).}
\]
\[
| t^* \text{ (priority loop), } t_k^{(w_w)} \text{ (random loop).}
\]
\[
| \overset{s}{\leftarrow} \text{ (raise), } ||t^* \overset{s}{\leftarrow} t^* \text{ (catch).}
\]
\[
| \overset{n}{\leftarrow} \overset{\setminus}{\leftarrow} t_i \text{ (priority).}, ||t_i^{(w)} \text{ (choice).}
\]
\[
| \overset{n}{\leftarrow} t_i \text{ (parallel).}
\]

This abstract syntax slightly differs from the concrete one on the following points.

(i) The empty behavior (e) and the empty behavior filter (t \triangleleft e) are internal constructs that will ease the definition of the semantics.

(ii) Random loops are normalized by making explicit their weight functions:

(a) the stop function \( \omega_s \) takes the number of already performed iterations and returns the relative weight of the “stop” choice;

(b) the continue function \( \omega_c \) takes the number of already performed iterations and returns the relative weight of the “continue” choice.

These functions are completely determined by the loop profile in the concrete program (interval or average, together with the corresponding static arguments). See Section 5.3 for a precise definition of these weight functions.

(iii) The number of already performed iterations \( k \) is syntactically attached to the loop; this is convenient to define the semantics in terms of rewriting (in the initial program, this number is obviously set to 0).
The predicate $e$ Satisfiability provide the two following procedures.

In order to generate constraints, the environment should (pseudo)random generation may vary from one implementation to another and they are not part of the reference abstract. As a matter of fact, resolution capabilities and random selection are also performed by the environment. We keep this environment resolution, weight evaluation, and random selection are also executed within an environment which stores the variable values (inputs and memories). Constraint resolution, weight evaluation, and random selection are also performed by the environment. We keep this environment abstract. As a matter of fact, resolution capabilities and (pseudo)random generation may vary from one implementation to another and they are not part of the reference semantics.

The semantics is given in terms of constraints generator. In order to generate constraints, the environment should provide the two following procedures.

**Satisfiability**

the predicate $e \models c$ is true if and only if the constraint $c$ is satisfiable in the environment $e$.

**Priority sort**

Executing choices first requires to evaluate the weights in the environment. This is possible (and straightforward) because weights may dynamically depend on uncontrollable variables (memories, inputs), but not on controllable variables (outputs, locals). Some weights may be evaluated to 0, in which case the corresponding choice is forbidden. Then a random selection is made, according to the actual weights, to determine a total order between the choices.

For instance, consider the following list of pairs (trace/weight), where $x$ and $y$ are uncontrollable variables,

\[
(t_1/x + y), (t_2/1), (t_3/y), (t_4/2).
\]

In an environment, where $x = 3$ and $y = 0$, weights are evaluated to

\[
(t_1/3), (t_2/1), (t_3/0), (t_4/2).
\]

The choice $t_3$ is erased and the remaining choices are randomly sorted according to their weights. The resulting (total) order may be

(i) $t_1, t_2, t_4$ with a probability $3/6 \times 1/3 = 1/6$,

(ii) $t_1, t_4, t_2$ with a probability $3/6 \times 2/3 = 1/3$,

(iii) $t_4, t_1, t_2$ with a probability $2/6 \times 3/4 = 1/4$,

(iv) so on.

All these treatments are “hidden” within the function $Sort$, which takes a list of pairs (choice/weights) and returns a totally ordered list of choices.

**4.3. The step function**

An execution step is performed by the function $\text{Step}(e, t)$ taking an environment $e$ and a trace expression $t$. It returns an action which is either

(i) a transition $\xleftarrow{c} n$, which means that $t$ produces a satisfiable constraint $c$ and rewrite itself in the (next) trace $n$,

(ii) a termination $\xleftarrow{x}$, where $x$ is a termination flag which is either $\epsilon$ (normal termination), $\delta$ (deadlock) or some user-defined exception.

**Definition 2.** $A$ denotes the set of actions and $X$ denotes the set of termination flags.

**4.4. The recursive step function**

The step function $\text{Step}(e, t)$ is defined via a recursive function $\delta_e(t, g, s)$, where the parameters $g$ and $s$ are continuation functions returning actions.

(i) $g : C \times T \rightarrow A$ is the goto function defining how a local transition should be treated according to the calling context.

(ii) $s : X \rightarrow A$ is the stop function defining how a local termination should be treated according to the calling context.

At the top-level, $\delta_e$ is called with the trivial continuations,

\[
\text{Step}(e, t) = \delta_e(t, g, s) \text{ with } g(c, v) = \xleftarrow{c} v, \quad s(x) = \xleftarrow{x}.
\]
Basic traces

The empty behavior raises the termination flag in the current context. A raise statement terminates with the corresponding flag. At last, a constraint generates a goto or raises a deadlock depending on its satisfiability:

\[
\delta_e(\varepsilon, g, s) = s(\varepsilon)
\]

\[
\delta_e(c, \varepsilon, g, s) = s(x)
\]

\[
\delta_e(c, g, s) = \text{if } (e \models c) \text{ then } g(c, \varepsilon) \text{ else } s(\delta).
\]

Sequence

The rule is straightforward;

\[
\delta_e(t \cdot t', g, s) = \delta_e(t, g', s'),
\]

where \(g'(c, n) = g(c, n \cdot t')\)

\[
s'(x) = \text{if } x = \varepsilon \text{ then } \delta_e(t', g, s) \text{ else } s(x).
\]

Priority choice

We only give the definition of the binary choice, since the operator is right-associative. This rule formalizes the reactivity principle. All possibilities in \(t\) must have failed before \(t'\) is taken into account,

\[
\delta_e(t > t', g, s) = \text{if } (r \neq \delta) \text{ then } r \text{ else } \delta_e(t', g, s),
\]

where \(r = \delta_e(t, g, s).\)

Empty filter and priority loop

The empty filter intercepts the termination of \(t\) and replaces by a deadlock,

\[
\delta_e(t \setminus \varepsilon, g, s) = \delta_e(t, g, s') \quad \text{where } s'(x) = \text{if } (x = \varepsilon) \text{ then } \delta_e(t', g, s) \text{ else } s(x).
\]

The semantics of the loop results from the equivalence

\[
t^* \iff (t \setminus \varepsilon) \cdot t^* > \varepsilon.
\]

Catch

This internal operator \(([n \leftarrow t]\)) covers the cases of try \((z = \delta)\) and catch \((z\) is a user-defined exception)

\[
\delta_e([t \leftarrow z, t'], g, s) = \delta_e(t, g', s'),
\]

where \(g'(c, n) = g(c, [n \leftarrow t'])\)

\[
s'(x) = \text{if } (x = z) \text{ then } \delta_e(t', g, s) \text{ else } s(x).
\]

Parallel composition

We only give the definition of the binary case, since the operator is right-associative;

\[
\delta_e(t \& t', g, s) = \delta_e(t, g', s'),
\]

where \(s'(x) = \text{if } (x = \varepsilon) \text{ then } \delta_e(t', g, s) \text{ else } s(x)\),

\[
g'(c, n) = \delta_e(t', g''', s'''),
\]

where \(s''(x) = \text{if } (x = \varepsilon) \text{ then } g(c, n) \text{ else } s(x)\)

\[
g'''(c', n') = g(c \land c', n \land n').
\]

Weighted choice

The evaluation of the weights and the (random) total ordering of the branches are both performed by the function \(\text{Sort}_e\) (cf., Section 4.2).

If \(\text{Sort}_e(\{t_i/w_i\}_{i=1}^n) = \emptyset : \delta_e(\{t_i/w_i\}_{i=1}^n, t_0/w_0, g, s) = s(\delta)\)

otherwise, \(\delta_e(\{t_i/w_i\}_{i=1}^n, t_0/w_0, g, s) = \delta_e(\setminus \text{Sort}_e(t_i/w_1, \ldots, t_n/w_n), g, s).\)

Random loop

We recall that this construct is labelled by two weight functions (\(\omega_e\) for continue, \(\omega_s\) for stop) and by the current number of already performed iterations \(i\). The weight functions are evaluated for \(i\) and the statement is then equivalent to a binary weighted choice,

\[
t_i^{(\omega_e, \omega_s)} \iff (t \setminus \varepsilon) \cdot t_i^{(\omega_e, \omega_s)}/\omega_e(i) \mid \varepsilon/\omega_s(i),
\]

Note that, the semantics follows the well-founded loop principle.

4.5. A complete execution

Solving a constraint

The main role of the environment is to store the values of uncontrollable variables; it is a pair of stores “past values, input values.” For such an environment \(e = (\rho, i)\) and a satisfiable constraint \(c\), we suppose given a procedure able to produce a particular solution of \(c : \text{Solve}_{\rho, i}(c) = y\) (where \(y\) is a store of controllable variables). We keep this Solve function abstract, since it may vary from one implementation to another (see Section 5).

Execution algorithm

A complete run is defined according to

(i) a given sequence of input stores \(I_0, I_1, \ldots, I_n\),
(ii) an initial (main) trace \(t_0\),
(iii) an initial previous store (values of pre variables) \(\rho_0\).
A run produces a sequence of (controllable variables) stores $y_1, y_2, \ldots, y_k$, where $k \leq n$. For defining this output sequence, we use intermediate sequences of traces $(t_1, \ldots, t_{k+1})$, previous stores $(\rho_1, \ldots, \rho_k)$, environments $(e_0, \ldots, e_k)$, and constraints $(c_0, \ldots, c_k)$. The relation between those sequences are listed below, for all step $j = 0 \cdots k$:

(i) the current environment is made of previous and input values, $e_j = (\rho_j, i_j)$,

(ii) the current trace makes a transition, $e_j : t_j \xrightarrow{c_j} t_{j+1}$,

(iii) a solution of the constraint is elected, $y_j = \text{Solve}_{c_j}(e_j)$,

(iv) the previous store for the next step is the union of current inputs/outputs: $\rho_{j+1} = (i_j \oplus y_j)$.

At the end, we have

(i) either $k = n$, which means that the execution has run to completion,

(ii) or $(\rho_{k+1}, t_{k+1}) : t_{k+1} \xrightarrow{x}$ which means that it has been aborted.

5. IMPLEMENTATION

A prototype has been developed in Ocaml. The constraint generator strictly implements the operational semantics presented in the previous section. The tool can do the following.

(i) Interpret/simulate Lutin programs in a file-to-file (or pipe-to-pipe) manner. This tool serves for simulation/protootyping; several Lutin simulation sessions can be combined with other reactive process in order to animate a complex system.

(ii) Compile Lutin programs into the internal format of the testing tool Lurette. This format, called Lucky, is based on flat, explicit automata [14]. In this case, Lutin serves as a high-level language for designing test scenarios.

5.1. Notes on constraint solvers

The core semantics only defines how constraints are generated, but not how they are solved. This choice is motivated by the fact that there is no “ideal” solver.

A required characteristic of such a solver is that it must provide a constructive, complete decision procedure; methods that can fail and/or that are not able to exhibit a particular solution are clearly not suitable. Basically, a constraint solver should provide the following.

(i) A syntactic analyzer for checking if the constraints are supported by the solver (e.g., linear arithmetics); this is necessary because the language syntax allows to write arbitrary constraints.

(ii) A decision procedure for the class of constraints accepted by the checker.

(iii) A precise definition of the election procedure which selects a particular solution (e.g., in terms of fairness).

Even with those restrictions, there is no obvious best solver as follows.

(i) It may be efficient, but limited in terms of capabilities.

(ii) It may be powerful, but likely to be very costly in terms of time and memory.

The idea is that the user may choose between several solvers (or several options of a same solver) the one which best fits his needs.

The solver that is currently used is presented in the next section.

5.2. The Boolean/linear constraint solver

Actually, we use the solver [15] that have been developed for the testing tool Lurette [16, 17]. This solver is quite powerful, since it covers Boolean algebra and linear arithmetics. Concretely, constraints are solved by generating a normalized representation mixing binary decision diagrams and convex polyhedra. This constraint solver is sketched below and fully described in [15]

First of all, each atomic numeric constraint (e.g., $x + y > 1$) is replaced by a fresh Boolean variable. Then, the resulting constraint is translated into a BDD. Figure 4 shows a graphical representation of a BDD; then (resp., else) branches are represented at the left-hand-side (resp., right-hand-side) of the tree. This BDD contains 3 paths to the true leaf: $ade$, $\overline{a}bc\overline{c}$, and $\overline{a}bd$. When we say that the monomial (conjunction of literals) $\overline{a}bc\overline{c}$ is a solution of the formula. It means that variables $a$ and $e$ should be true; variables $b$ and $c$ should be true; and variable $d$ can be either true or false. The monomial $\overline{a}bc\overline{c}$, therefore, represents two solutions, whereas $ade$ and $\overline{a}bd$ represents 4 solutions each, since 2 variables are left unconstrained.

In Figure 4 and in the following, for the sake of simplicity, we draw trees instead of DAGs. The key reason why BDDs work well in practice is that in their implementations, common subtrees are shared. For example, only one node “true” would be necessary in that graph. Anyway, the algorithms work on DAGs the same way as they work on trees.

Random choice of Boolean values

The first step consists in selecting a Boolean solution. Once the constraint has been translated into a BDD, we have a (hopefully compact) representation of the set of solutions.
We first need to randomly choose a path into the BDD that leads to a true leaf. But if we naively performed a fair toss at each branch of the BDD during this traversal, we would be very unfair. Indeed, consider the BDD of Figure 4; the monomial ade has 50% of chances to be tried, whereas abcd and abcd have 25% each. One can easily imagine situations where the situation is even worse. This is the reason why counting the solutions before drawing them is necessary.

Once each branch of the BDD is decorated with its solution number performing a fair choice among Boolean solutions is straightforward.

Random choice of numeric values

From the BDD point of view, numeric constraints are just Boolean variables. Therefore, we have to know if the obtained set of atomic numeric constraints is satisfiable. For that purpose, we use a convex polyhedron library [18].

However, a solution from the logical variables point of view may lead to an empty set of solutions for numeric variables. In order to chose a Boolean monomial that is valid with respect to numerics, a (inefficient) method would be to select at random a path in the BDD until that selection corresponds to a satisfiable problem for the numeric constraints. The actual algorithm is more sophisticated [15], but the resulting solution is the same.

When there are solutions to the set of numeric constraints, the convex polyhedron library returns a set of generators (the vertices of the polyhedron representing the set of solutions). Using those generators, it is quite easy to choose point inside (or more interestingly, at edges or at vertices) the polyhedron.

Using polyhedra is very powerful, but also very costly. However the solver benefits from several years of experimentation and optimizations (partitioning, switch from polyhedra to intervals, whenever it is possible).

5.3. Notes on predefined loop profiles

In the operational semantics, loops with iteration profile are translated into binary weighted choices. Those weights are dynamic; they depend on the number of (already) performed iterations k.

Interval loops

For the “interval” profile, those weights functions are formally defined and thus, they could take place in the reference semantics of the language. For a given pair of integers (min, max) such that 0 ≤ min ≤ max and a number k of already performed iterations, we have the following:

(i) if k < min, then \(\omega_i(k) = 0\) and \(\omega_c(k) = 1\) (loop is mandatory);
(ii) if k ≥ max, then \(\omega_i(k) = 1\) and \(\omega_c(k) = 0\) (stop is mandatory);
(iii) if min ≤ k < max, then \(\omega_i(k) = 1\) and \(\omega_c(k) = 1 + \max - k\).

Average loops

There is no obvious solution for implementing the “average” profile in terms of weights. A more or less sophisticated (and accurate) solution should be retained, depending on the expected precision.

In the actual implementation, for an average value \(av\) and a standard variation \(sv\), we use a relatively simple approximation as follows.

(i) First of all, the underlying discrete repartition law is approximated by a continuous (Gaussian) law. As a consequence, the result will not be accurate if \(av\) is too close to 0 and/or if \(sv\) is too big comparing to \(av\). Concretely, we must have \(10 < 4 \times sv < av\).

(ii) It is well known that no algebraic definition for the Gaussian repartition function exists. This function is then classically approximated by using an interpolation table (512 samples with a fixed precision of 4 digits).

6. CONCLUSION

We propose a language for describing constrained-random reactive systems. Its first purpose is to describe test scenarios, but it may also be useful for prototyping and simulation.

We have developed a compiler/interpreter which strictly implements the operational semantics presented here. Thanks to this tool, the language is integrated into the framework of the Lurette tool, where it is used to describe test scenarios. Further works concerns the integration of the language within a more general prototyping framework.

Other works concern the evolution of the language. We plan to introduce a notion of signal (i.e., event) which is useful for describing values that are not always available (this is related to the notion of clocks in synchronous languages). We also plan to allow the definition of (mutually) tail-recursive traces. Concretely, that means that a new programming style would be allowed, based on explicit concurrent, hierarchic automata.

REFERENCES


EURASIP Journal on Embedded Systems


The recently proposed reactive processing architectures are characterized by instruction set architectures (ISAs) that directly support reactive control flow including concurrency and preemption. These architectures provide efficient execution platforms for reactive synchronous programs; however, they do require novel compiler technologies, notably with respect to the handling of concurrency. Another key quality of the reactive architectures is that they have very predictable timing properties, which make it feasible to analyze their worst-case reaction time (WCRT). We present an approach to compile programs written in the synchronous language Esterel onto a reactive processing architecture that handles concurrency via priority-based multithreading. Building on this compilation approach, we also present a procedure for statically determining tight, safe upper bounds on the WCRT. Experimental results indicate the practicality of this approach, with WCRT estimates to be accurate within 22% on average.

Copyright © 2008 Marian Boldt et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

The programming language Esterel [1] has been designed for developing control-dominated reactive software or hardware systems. It belongs to the family of synchronous languages [2], which have a formal semantics that abstracts away runtime uncertainties, and allow abstract, well-defined, and executable descriptions of the application at the system level. Hence these languages are particularly suited to the design of safety-critical real-time systems. To express reactive behavior, Esterel offers numerous powerful control flow primitives, in particular concurrency and various preemption operators. Concurrent threads can communicate back and forth instantaneously, with a tight semantics that guarantees deterministic behavior. This is valuable for the designer, but also poses implementation challenges.

Besides being compiled to C and executed as software, or being compiled to VHDL and synthesized to hardware, Esterel can be executed on a reactive processor [3]. These processors directly support reactive control flow, such as preemption and concurrency, in their instruction set architecture (ISA). One approach to handle concurrency is multithreading, as implemented in the Kiel Esterel processor (KEP). The KEP uses a priority-based scheduler, which makes threads responsible to manage their own priorities. This scheme allows to keep the scheduler very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing a thread’s priority costs an instruction. One challenge for the compiler is to compute these priorities in a way that on the one hand preserves the execution semantics of Esterel and on the other hand does not lead to too many changes of the priorities, since this would decrease the execution speed. We have developed a priority assignment algorithm that makes use of a special concurrent control flow graph and has a complexity that is linear in the size of that graph, which in practice tends to be linear in the size of the program.

Apart from efficiency concerns, which may have been the primary driver towards reactive processing architectures, one of their advantages is their timing predictability. To leverage this, we have augmented our compiler with a timing analysis capability. As we here are investigating the timing behavior for reactive systems, we are specifically concerned with computing the maximal time it takes to compute a single reaction. We refer to this time, which is the time from given input events to generated output events, as worst-case reaction time (WCRT). The WCRT determines the maximal rate for the interaction with the environment.
There are two main factors that facilitate the WCRT analysis in the reactive processing context. These are on the one hand the synchronous execution model of Esterel, and on the other hand the direct implementation of this execution model on a reactive processor. Furthermore, these processors are not designed to optimize (average) performance for general purpose computations, and hence do not have a hierarchy of caches, pipelines, branch predictors, and so forth. This leads to a simpler design and execution behavior and further facilitates WCRT analysis. Furthermore, there are reactive processors, such as the KEP, which allow to fix the reaction lengths to a predetermined number of clock cycles, irrespective of the number of instructions required to compute a specific reaction, in order to minimize the jitter.

We here present a WCRT analysis of complete Esterel programs including concurrency and preemption. The analysis computes the WCRT in terms of KEP instruction cycles, which roughly match the number of executed Esterel statements. As part of the WCRT analysis, we also present an approach to calculate potential instantaneous paths, which may be used in compiler analysis and optimizations that go beyond WCRT analysis.

Thus this paper is concerned with both the compilation and the timing analysis of Esterel programs executed on multithreaded reactive processors. Previous reports presented earlier results in both fields [4, 5]. This paper extends and updates these reports, and represents the first comprehensive description of these two closely interrelated areas. Further details can be found in the theses of the first author [6, 7].

In the following section, we consider related work. In Section 3, we will give an introduction into the synchronous model of computation for Esterel and the KEP. We outline the generation of a concurrent KEP assembler graph (CKAG), an intermediate graph representation of an Esterel program, which we use for our analysis. Section 4 explains the compilation and Section 5 represents the algorithm for the WCRT analysis. Section 6 presents experimental results that compare the WCRT estimates with values obtained from exhaustive simulation. The paper concludes in Section 7.

2. RELATED WORK

In the past, various techniques have been developed to synthesize Esterel into software; see Potop-Butucaru et al. [8] for an overview. The compiler presented here belongs to the family of simulation-based approaches, which try to emulate the control logic of the original Esterel program directly, and generally achieve compact and yet fairly efficient code. These approaches first translate an Esterel program into some specific graph formalism that represents computations and dependencies, and then generate code that schedules computations accordingly. The EC/Synopsys compiler first constructs a concurrent control flow graph (CCFG), which it then sequentializes [9]. Threads are statically interleaved according to signal dependencies, with the potential drawback of superfluous context switches; furthermore, code sections may be duplicated if they are reachable from different control points. The SAXO-RT compiler [10] divides the Esterel program into basic blocks, which schedule each other within the current and subsequent logical tick. An advantage relative to the Synopsis compiler is that the SAXO-RT compiler does not perform unnecessary context switches and largely avoids code duplications; however, the scheduler it employs has an overhead proportional to the total number of basic blocks present in the program. The gcc2c compiler [11] is based on the graph code (GRC) format, which preserves the state-structure of the given program and uses static analysis techniques to determine redundancies in the activation patterns. A variant of the GRC has also been used in the Columbia Esterel compiler (CEC) [12], which follows SAXO-RT’s approach of dividing the Esterel program into atomically executed basic blocks. However, their scheduler does not traverse a score board that keeps track of all basic blocks, but instead uses a compact encoding based on linked lists, which has an overhead proportional to just the number of blocks actually executed.

In summary, there is currently not a single Esterel compiler that produces the best code on all benchmarks, and there is certainly still room for improvements. For example, the simulation-based approaches presented so far restrict themselves to interleaved single-pass thread execution, which in the case of repeated computations (“schizophrenia” [13]) requires code replications.

We differ from these approaches in that we do not want to compile Esterel to C, but instead want to map it to a concurrent reactive processing ISA. Initial reactive ISAs did not consider full concurrency [14, 15] and will not be discussed further here. Since then, two alternatives have been proposed that do include concurrency, namely multiprocessing and multithreading.

The multiprocessing approach is represented by the EMPEROR [16], which uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential for execution speed-ups relative to single-processor implementations. However, their execution model potentially requires to replicate parts of the control logic at each processor. The EMPEROR Esterel compiler 2 (EEC2) [16] is based on a variant of the GRC, and appears to be competitive even for sequential executions on a traditional processor. However, their synchronization mechanism, which is based on a three-valued signal logic, does not seem able to take compile-time scheduling knowledge into account, and instead repeatedly cycles through all threads until all signal values have been determined.

The multithreading approach has been introduced by the Kiel Esterel processor family and has subsequently been adapted by the STARPro architecture [17], a successor of the EMPEROR. The compilation for this type of architecture is a subject of this paper. In some sense, compilation onto KEP assembler is relatively simple, due to the similarities between the Esterel and the KEP assembler. However, we do have to compute priorities for the scheduling mechanism of the KEP, and cannot hard-code the scheduling-mechanism into the generated code directly. Incidentally, it is this dynamic, hardware-supported scheduling that contributes to the efficiency of the reactive processing approach.
It has also been proposed to run Esterel programs on a virtual machine (BAL [18]), which allows a very compact byte code representation. In a way, this execution platform can be considered as an intermediate form between traditional software synthesis and reactive processing; it is a software running on traditional processors, but uses a more abstract instruction set. The proposal by Plummer et al. also uses a multithreaded concurrency model, as in the KEP platform considered here. However, they do not assume the existence of a run-time scheduler, but instead hand control explicitly over between threads. Thus their scheduling problem is related to ours, but does not involve the need to compute priorities as we have to do here. Instead, they have to insert explicit points for context switches. The main difference in both approaches is that the KEP only switches to active threads, while the BAL switches to statically defined control points. One could, however, envision a virtual machine that has an ISA that adopts our multithreading model (a straightforward, albeit inefficient VM would be a KEP simulator), and for which the approach presented here could be applied.

One of the byproducts of our compilation approach is dead code elimination (DCE), see also Section 4.3. Our approach here is rather conservative, considering only static reachability. A more aggressive approach to DCE based on Esterel* (an extension of Esterel with a noninstantaneous jump instruction) has been presented by Tardieu and Edwards [19]. Their approach, as well as other work that performs reachability analysis as part of constructiveness analysis [20], is more involved than our approach in that they perform an (more or less conservative) analysis of the reachable state space.

Regarding timing analysis, there exist numerous approaches to classical worst-case execution time (WCET) analysis. For surveys see, for example, Puschner and Burns [21] or Wilhelm et al. [22]. These approaches usually consider (subsets) of general purpose languages, such as C, and take information on the processor designs and caches into account. It has long been established that to perform an exact WCET analysis with traditional programming languages on traditional processors is difficult, and in general not possible for Turing-complete languages. Therefore WCET analysis typically impose fairly strong restrictions on the analyzed code, such as a-priori known upper bounds on loop iteration counts, and even then control flow analysis is often overly conservative [23, 24]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict how much time exactly the execution of these instructions consumes, due to pipelining, out-of-order execution, argument-dependent execution times (e.g., particularly fast multiply-by-zero), and caching of instructions and/or data [25]. Finally, if external interrupts are possible or if an operating system is used, it becomes even more difficult to predict how long it really takes for an embedded system to react to its environment. Despite the advances already made in the field of WCET analysis, it appears that most practitioners today still resort to extensive testing plus adding a safety margin to validate timing characteristics.

To summarize, performing conservative yet tight WCET analysis appears by no means trivial and is still an active research area.

Whether WCRT can be formulated as a classical WCET problem or not depends on the implementation approach. If the implementation is based on sequentialization such that there exist two dedicated points of control at the beginning and the end of each reaction, respectively, then WCRT can be formulated as WCET problem; this is the case, for example, if one “automaton function” is synthesized, which is called during each reaction. If, however, the implementation builds on a concurrent model of execution, where each thread maintains its own state-of-control across reactions, then WCRT requires not only determining the maximal length of predefined instruction sequences, as in WCET, but one also has to analyze the possible control point pairs that delimit these sequences. Thus, WCRT is more elementary than WCET in the sense that it considers single reactions, instead of whole programs, and at the same time WCRT is more general than WCET in that it is not limited to predefined control boundaries.

One step to make the timing analysis of reactive applications more feasible is to choose a programming language that provides direct, predictable support for reactive control flow patterns. We argue that synchronous languages, such as Esterel, are generally very suitable candidates for this, even though there has been little systematic treatment of this aspect of synchronous languages so far. One argument is that synchronous languages naturally provide a timing granularity at the application level, the logical ticks that correspond to system reactions, and impose clear restriction onto what programs may do within these ticks. For example, Esterel has the rule that there cannot be instantaneous loops: within a loop body, each statically feasible path must contain at least one tick-delimiting instruction, and the compiler must be able to verify this. Another argument is that synchronous languages directly express reactive control flow, including concurrency, thus lowering the need for an operating system with unpredictable timing.

Logothetis et al. [26, 27] have employed model checking to perform a precise WCET analysis for the synchronous language Quartz, which is closely related to Esterel. However, their problem formulation was different from the WCRT analysis problem we are addressing. They were interested in computing the number of ticks required to perform a certain computation, such as a primality test, which we would actually consider to be a transformational system rather than a reactive system [28]. We here instead are interested in how long it may take to compute a single tick, which can be considered an orthogonal issue.

Ringler [29] considers the WCET analysis of C code generated from Esterel. However, his approach is only feasible for the generation of circuit code [13], which scales well for large applications, but tends to be slower than the simulation-based approach.

Li et al. [15] compute a of sequential Esterel programs directly on the source code. However, they did not address concurrency, and their source-level approach could not
consider compiler optimizations. We perform the analysis on an intermediate level after the compilation, as a last step before the generation of assembler code. This also allows a finer analysis and decreases the time needed for the analysis.

One important problem that must be solved when performing WCRT analysis for Esterel is to determine whether a code segment is reachable instantaneously, or delayed, or both. This is related to the well-studied property of surface and depth of an Esterel program, that is, to determine whether a statement is instantaneously reachable or not, which is also important for schizophrenic Esterel programs [13]. This was addressed in detail by Tardieu and de Simone [30]. They also point out that an exact analysis of instantaneous reachability has NP complexity. We, however, are not only interested whether a statement can be instantaneous, but also whether it can be noninstantaneous.

3. ESTEREL, THE KIEL ESTEREL PROCESSOR AND THE CONCURRENT KEP ASSEMBLER GRAPH

Next we give a short overview of Esterel and the KEP. We also introduce the CKAG, a graph-representation of Esterel, which is used both for the compilation and the WCRT analysis.

3.1. Esterel

The execution of an Esterel program is divided into logical instants, or ticks, and communication within or across threads occurs via signals. At each tick, a signal is either present (emitted) or absent (not emitted). Esterel statements are either transient, in which case they do not consume logical time, or delayed, in which case execution is finished for the current tick. Per default statements are transient, and these include for example emit, loop, present, or the preemption operators. Delayed statements include pause, (nonimmediate) await, and every. Esterel’s parallel operator, ||, groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated.

Esterel offers two types of preemption constructs. An abortion kills its body when an abortion trigger occurs. We distinguish strong abortion, which kills its body immediately (at the beginning of a tick), and weak abortion, which lets its body receive control for a last time (abortion at the end of the tick). A suspension freezes the state of a body in the instant when the trigger event occurs.

Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is declared with a trap scope, and is thrown (raised) with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are further rules when traps are nested or when the trap scope includes concurrent threads. If one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent threads are weakly aborted; if concurrent threads execute multiple exit instructions in the same tick, the outermost trap takes priority.

3.1.1. Examples

As an example of a simple, nonconcurrent program consider the module ExSeq shown in Figure 1(a). As the sample execution trace illustrates, the module emits signal R in every instant, until it is aborted by the presence of the input signal I. As this is a weak abortion, the abortion body gets to execute (emit R) one last time when it is aborted, followed by an emission of S.

The program ExPar shown in Figure 2(a) introduces concurrency: a thread that emits R and then terminates, and a concurrent thread that emits S, pauses for an instant, emits T, and then terminates are executed in an infinite loop. During each loop iteration, the parallel terminates when both threads have terminated, after which the subsequent loop iteration is started instantaneously, that is, within the same tick.

A slightly more involved example is the program Edwards02 [9, 10], shown in Figure 3(a). This program implements the following behavior: whenever the signal S is present, (re-)start two concurrent threads. The first thread first awaits a signal I; it then continuously emits R until A is present, in which case it emits R one last time (weak abortion of the sustain), emits O, and terminates. The second thread tests every other tick for the presence of R, in which case it emits A.

3.1.2. Statement dismantling

At the Esterel level, one distinguishes kernel statements and derived statements. The derived statements are basically syntactic sugar, built up from the kernel statements. In principle, any set of Esterel statements from which the remaining statements can be constructed can be considered a valid set of kernel statements, and the accepted set of Esterel kernel statements has evolved over time. For example, the halt statement used to be considered a kernel statement, but is now considered to be derived from loop and pause. We here adopt the definition of which statements are kernel statements from the v5 standard [31]. The process of expanding derived statements into equivalent, more primitive statements—which may or may not be kernel statements—is also called dismantling. The Esterel program Edwards02-dism (Figure 3(b)) is a dismantled version of Edwards02. It is instructive to compare this program to the original, undismantled version.

3.2. The Kiel Esterel processor

The instruction set architecture (ISA) of the KEP is very similar to the Esterel language. Part of the KEP instruction set is shown in Table 1; a complete description can be found elsewhere [32]. The KEP instruction set includes all kernel statements (see Section 3.1.2), and in addition some frequently used derived statements. The KEP ISA also includes valued signals, which cannot be reduced to kernel statements. The only parts of Esterel v5 that are not part of the KEP ISA are combined-signal handling and external-task handling, as they both seem to be used only rarely in
Figure 1: A sequential Esterel example. The body of the KEP assembler program (without interface declaration and initialization of the TickManager) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles.

Figure 2: A concurrent example program.
sample traces are shown in Figures 1(c)-1(d) and 2(c)-2(d), as assembler programs corresponding to the KEP to execute arbitrary Esterel programs. The KEP known translations into kernel statements exist, allowing instruction cycle. For more complicated statements, well-practice. However, adding these capabilities to the KEP ISA seems relatively straightforward.

Due to this direct mapping from Esterel to the KEP ISA, most Esterel statements can be executed in just one instruction cycle. For more complicated statements, well-known translations into kernel statements exist, allowing the KEP to execute arbitrary Esterel programs. The KEP assembler programs corresponding to ExSeq and ExPar and sample traces are shown in Figures 1(c)-1(d) and 2(c)-2(d), respectively, and the KEP assembler program for Edwards02 is shown in Figure 3(c), respectively. Note that PAUSE is executed for at least two consecutive ticks, and consumes an instruction cycle at each tick.

The KEP provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body [33]. Therefore, no additional instruction cycles are needed to test for preemption during each tick. Only upon entering a preemption scope two cycles are needed to initialize the Watcher, as for example the WABORTI instruction in ExSeq (Figure 1(c)). To aid readability, we here use the convention of subscripting KEP instructions with the line number where they occur.

To implement concurrency, the KEP employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses, thread id and dynamically changing priorities: between all active threads, the thread with the highest priority is scheduled. If there is more than one thread with this priority, the highest thread id wins. The scheduler is very lightweight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing a thread’s priority costs an instruction. The priority-based execution scheme allows on the one hand to enforce an ordering among threads that obeys the constraints given by Esterel’s semantics, but on the other hand avoids unnecessary context switches. If a thread lowers its priority during execution but still has the highest priority, it simply keeps executing.

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into KEP assembler as follows. First, threads are forked by a series of instructions that consist of n PAR instructions and one PARE instruction. Each PAR instruction creates one thread, by assigning a nonnegative priority, a start address, and the thread id. The end address of the thread is either given implicitly by the start address specified in a subsequent PAR instruction, or, if
Table 1: Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

<table>
<thead>
<tr>
<th>Mnemonic, operands</th>
<th>Esterel syntax</th>
<th>Cycles</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAR ( \text{prio}_1, \text{startAddr}_1, \text{id}_1 )</td>
<td>[ \text{PAR }[\text{startAddr}_1, \text{id}_1] ]</td>
<td>( n + 1 )</td>
<td>For each thread, one PAR is needed to define the start address, thread id, and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is denoted via PARE.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PAR ( \text{prio}_n, \text{startAddr}_n, \text{id}_n )</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PARE endAddr \text{startAddr}_i: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>startAddr: [ p_1 \parallel \cdots \parallel p_n ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>startAddr: [ p_1 \parallel \cdots \parallel p_n ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>endAddr: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>JOIN</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PRIO \text{prio} ]</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[W]ABORT[\text{startAddr}, \text{endAddr}:]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{abort} )</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>endAddr: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SUSPEND[\text{startAddr}, \text{endAddr}:]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>suspend</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>endAddr: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>startAddr: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>trap ( \text{in} )</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>EXIT \text{exitAddr} \text{startAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exit ( \text{in} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exitAddr: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PAUSE ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>pause</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>AWAiT [ \text{startAddr}, \text{endAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{wait} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SIGNAL [ \text{startAddr}, \text{endAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{signal} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>EMIT [ \text{startAddr}, \text{endAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{emit} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SUSTAIN [ \text{startAddr}, \text{endAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{sustain} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PRESENT [ \text{startAddr}, \text{endAddr}: ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{present} )</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>HALT</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>halt</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>addr: \ldots \text{GOTO addr} ]</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>loop \ldots \text{end loop}</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

There is no more thread to be created, it is specified in a PARE instruction. The code block for the last thread is followed by a JOIN instruction, which waits for the termination of all forked threads and concludes the concurrent statement. The example in Figure 2(c) illustrates this: instruction L4 constitutes thread 1, thread 2 spans L5–L8, and the remaining instructions belong to the main thread, which implicitly has id 0.

The priority of a thread is assigned when the thread is created (with the aforementioned PAR instruction), and can be changed subsequently by executing a priority setting instruction (PRIO). A thread keeps its priority across delay instructions; that is, at the start of a tick it resumes execution with the priority it had at the end of the previous tick. This mechanism allows an arbitrary interleaving of thread execution for communicating among threads within the same logical tick. Therefore, a thread may be executed partially, then control may jump to another thread, and later return to the first thread, all within the same tick.

When a concurrent statement terminates, through regular termination of all concurrent threads or via an exception/abort, the priorities associated with the terminated threads also disappear, and the priority of the main thread is restored to the priority upon entering the concurrent statement.

The KEP contains a TickManager, which monitors how many instructions are executed in the current logical tick. To minimize jitter, a maximum number of instructions for each logical tick can be specified, via the “special” valued signal _TICKLEN_. If the current tick needs less instructions, the start of the next tick is delayed, making the maximum number of instructions the exact number of instructions. If the tick needs more instructions, an error-output is set. Hence a tight, but conservative upper bound of the maximal
instructions for one tick, as computed by the WCRT analysis presented in Section 5, is of direct value for the KEP. See Li et al. [15] for details on the TickManager and the relation between the maximum number of instruction per logical tick and the physical timing constraints from the environment perspective.

Note that the KEP compiler per default computes a value for the WCRT and adds a corresponding assembler instruction that specifies a value for \_TICKLEN. However, the KEP does not require such a specification of \_TICKLEN. If \_TICKLEN is left unspecified, the processor "runs freely" and starts the next logical tick as soon as the current tick is finished. This lowers, on average, the reaction time, at the price of a possible jitter.

3.3. The concurrent KEP assembler graph

The CKAG is a directed graph composed of various types of nodes and edges to match KEP program behavior. It is used during compilation from Esterel to KEP assembler, for, for example, priority assigning, dead code elimination, further optimizations, and the WCRT analysis. The CKAG is generated from the Esterel program via a simple structural translation. The only nontrivial aspect is the determination of noninstantaneous paths, which is needed for certain edge types. For convenience, we label nodes with KEP instructions; however, we could alternatively have used Esterel instructions as well.

The CKAG distinguishes the following sets of nodes, see also Figure 4:

- \( L \): label nodes (ellipses);
- \( T \): transient nodes (rectangles), which include \texttt{EMIT}, \texttt{PRESENT}, and so forth;
- \( D \): delay nodes (octagons), which correspond to delayed KEP instructions (\texttt{PAUSE}, \texttt{AWAIT}, \texttt{HALT}, \texttt{SUSTAIN});
- \( F \): fork nodes (triangles), corresponding to \texttt{PAR/PARE};
- \( J \): join nodes (inverted triangles), corresponding to \texttt{JOIN};
- \( N \): set of all nodes, with \( N = T \cup L \cup D \cup F \cup J \).

We also define

- \( A \): the abort nodes, which denote abortion scopes and correspond to [W]ABORT and \texttt{SUSPEND}; note that \( A \subseteq T \).

For each fork node \( n \ (n \in F) \), we define

\( n.join \): the \texttt{JOIN} statement corresponding to \( n \ (n.join \in J) \), and

\( n.sub \): the transitive closure of nodes in threads spawned by \( n \).

For abort nodes \( n \ (n \in A) \), we define

\( n.end \): the end of the abort scope opened by \( n \), and
\( n.scope \): the nodes within \( n \)'s abort scope.

A nontrivial task when defining the CKAG structure is to properly distinguish the different types of possible control flow, in particular with respect to their timing properties (instantaneous or delayed). We define the following types of successors for each \( n \):

- \( n.suc_c \): the control successors. These are the nodes that follow sequentially after \( n \), considering normal control flow without any abstractions. For \( n \in F \), \( n.suc_c \) includes the nodes corresponding to the beginnings of the forked threads.

The successors are statically inserted, based on the syntax of the Esterel program, based on the actual behavior, some of these can be removed. If \( n \) is the last node of a concurrent thread, \( n.suc_c \) includes the node for the corresponding \texttt{JOIN}—unless \( n \)'s thread is instantaneous and has a (provably) noninstantaneous sibling thread. Furthermore, the control successors exclude those reached via a preemption \((n.suc_w, n.suc_c)\)—unless \( n \) is an immediate strong abortion node, in which case \( n.end \in n.suc_c \).

- \( n.suc_w \): the weak abort successors. If \( n \in D \), this is the set of nodes to which control can be transferred immediately, that is when entering \( n \) at the end of a tick, via a weak abort; if \( n \) exits a \texttt{trap}, then \( n.suc_w \) contains the end of the \texttt{trap} scope; otherwise it is \( \emptyset \).

If \( n \in D \) and \( n \in m.scope \) for some abort node \( m \), it is \( m.end \in n.suc_w \) in case of a weak immediate abort, or in case of a weak abort if there can (possibly) be a delay between \( m \) and \( n \).

- \( n.suc_s \): the strong abort successors. If \( n \in D \), these are the nodes to which control can be transferred after a delay, that is when restarting \( n \) at the beginning of a tick, via a strong abort; otherwise it is \( \emptyset \).

If \( n \in D \) and \( n \in m.scope \) for some strong abort node \( m \), it is \( m.end \in n.suc_s \).

Note that this is not a delayed abort in the sense that an abort signal in one tick triggers the preemption in the next tick. Instead, this means that first a delay has to elapse, and
the abort signal must be present at the next tick (relative to the tick when \( n \) is entered) for the preemption to take place.

\( n.\text{suc}_e \): the exit successors. These are the nodes that can be reached by raising an exception.

\( n.\text{suc}_f \): the flow successors. This is the set \( n.\text{suc}_c \cup n.\text{suc}_w \cup n.\text{suc}_c \).

For \( n \in F \), we also define two kinds of fork abort successors. These serve to ensure a correct priority assignment to parent threads in case there is an abort out of a concurrent statement.

\( n.\text{suc}_{wf} \): the weak fork abort successors. This is the union of \( m.\text{suc}_w \setminus n.\text{sub} \) for all \( m \in n.\text{sub} \) where there exists an instantaneous path from \( n \) to \( m \).

\( n.\text{suc}_{sf} \): the strong fork abort successors. This is the set \( \{ (m.\text{suc}_w \cup m.\text{suc}_c) \setminus n.\text{sub} \mid m \in n.\text{sub} \} \setminus n.\text{suc}_{wf} \).

In the graphical representation, control successors are shown by solid lines, all other successors by dashed lines, annotated with the kind of successor.

The CKAG is built from Esterel source by traversing recursively over its abstract syntax tree (AST) generated by the Colombia Esterel compiler (CEC) [34]. Visiting an Esterel statement results in creating the according CKAG node. A node typically contains exactly one statement, except label nodes containing just address labels and fork nodes containing one PAR statement for each child thread initialization and a PAR statement. When a delay node is created, additional preemption edges are added according to the abortion/exception context.

Note that some of the successor sets defined above cannot be determined precisely by the compiler, but have to be (conservatively) approximated instead. This applies in particular to those successor types that depend on the existence of an instantaneous path. Here it may be the case that for some pair of nodes there does not exist such an instantaneous path, but that the compiler is not able to determine that. In such cases, the compiler conservatively assumes that there may be such an instantaneous path. This is a common limitation of Esterel compilers, and compilers differ in their analysis capabilities here—see also Section 4.1.

### 4. THE KEP COMPILER

A central problem for compiling Esterel onto the KEP is the need to manage thread priorities during their creation and their further execution. In the KEP setting, this is not merely a question of efficiency or of meeting given deadlines, but a question of correct execution. Specifically, we have to schedule threads in such a fashion that all signal dependencies are obeyed. Such dependencies arise whenever a signal is possible emitted and tested in the same tick; we must ensure that all potential emitters for a signal have executed before that signal is tested.

A consequence of Esterel’s synchronous model of execution is that there may be dependency cycles, which involve concurrent threads communicating back and forth within one tick. Such dependency cycles must be broken, for example, by a delay node, because otherwise it would not be possible for the compiler to devise a valid execution schedule that obeys all ordering (causality) constraints. In the Edwards02 example (Figure 3(a)), there is one dependency cycle, from the sustain \( R_9 \) instruction in the first parallel thread to the present \( R_9 \) in the second parallel to the emit \( A_7 \) back to the sustain \( R_9 \), which is weakly aborted whenever \( A \) is present. The dependency cycle is broken in the dismantled version, as there the sustain \( R \) has been separated into signal emission (emit \( R_9 \)) and a delay (pause), enclosed in a loop. The broken dependency cycle can also be observed in the CKAG, shown in Figure 5. Referring to nodes by the corresponding line numbers (the “Lxx” part of the node labels) in the KEP assembler code (Figure 3(c)), the cycle is \( L14 \rightarrow L23 \rightarrow L24 \rightarrow L17 \rightarrow L18 \rightarrow L14 \) it is broken by the delay in L17.

The priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. Due to the nonlinear control flow, it is still possible that a given statement may be executed with varying priorities. In principle, the architecture would therefore allow a fully dynamic scheduling. However, we here assume that the given Esterel program can be executed with a statically determined schedule, which requires the existence of no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [35] and the CEC compilers; see also Section 3.3. Note that there are also Esterel programs that are causally correct (constructive [1]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [36], which can then be translated into KEP assembler. Hence, the actual run-time schedule of a concurrent program running on KEP is static in the sense that if two statements that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each statement is known in advance. However, the run-time schedule is dynamic in the sense that due to the nonlinear control flow and the independent advancement of each program counter, it in general cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions. Instead, a run-time scheduling mechanism is needed that manages the interleaving according to the priority and actual program counter of each active thread.

To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each tick, all enabled threads are activated, and are subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction, in which case it keeps that new priority until it executes yet another PRIO instruction, and so on. Neither the scheduler nor other threads can change a thread’s priority. Note also that a PRIO instruction is considered instantaneous. The only noninstantaneous instructions, which delimit the logical
ticks and are also referred to delayed instructions, are the PAUSE instruction and derived instructions, such as AWAIT and SUSTAIN. This mechanism has a couple of implications.

(i) At the start of a tick, a thread is resumed with the priority corresponding to the last PRIO instruction it executed during the preceding ticks, or with the priority it has been created with if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it is not sufficient to execute
a PRIO instruction at the beginning of that tick; instead, we must already have executed that PRIO instruction in the preceding tick.

(ii) A thread is executed only if no other active thread has a higher priority. Once a thread is executing, it continues until a delayed statement is reached, or until its priority is lower than that of another active thread or equal to that of another thread with higher id. While a thread is executing, it is not possible for other inactive threads to become active; furthermore, while a thread is executing, it is not possible for other threads to change their priority. Hence, the only way for a thread’s priority to become lower than that of other active threads is to execute a PRIO instruction that lowers its priority below that of other active threads.

4.1. Annotating the CKAG with dependencies

In order to compute the thread priorities, we annotate the with additional information about already known priorities and dependencies. For all nodes \( n \), we define

\[
 n.\text{prio}: \text{the priority that the thread executing } n \text{ should be running with.}
\]

For \( n \in D \cup F \), we also define

\[
 n.\text{prionext}: \text{the priority that the thread executing } n \text{ should be resumed with in the subsequent tick.}
\]

We annotate each node \( n \) with the set of nodes that read a signal which is emitted by \( n \). It turns out that analogously to the distinction between \( \text{prio} \) and \( \text{prionext} \), we must distinguish between dependencies that affect the current tick and the next tick:

\[
 n.\text{dep}_i: \text{the dependency sinks with respect to } n \text{ at the current tick (the immediate dependencies)},
\]

\[
 n.\text{dep}_d: \text{the dependency sinks with respect to } n \text{ at the next tick (the delayed dependencies)}. 
\]

We here assume that the Esterel program given to our compiler has already been established to be causal (constructive), using one of the established constructiveness analysis procedures [20], as for example implemented in the Esterel v5 compiler. We therefore consider only dependencies that cross thread boundaries, as dependencies within a thread do not affect the scheduling. In other words, we assume that intrathread dependencies are already covered by control dependencies; would that not be the case, the program would not be causal, and should be rejected. Should we not want to rely on a separate constructiveness analysis, we would have to consider intrathread dependencies as well.

In general, dependencies are immediate, meaning that they involve statements that are entered at the same tick. An exception are dependencies between emissions of a strong abort trigger signal and corresponding delay nodes within the abort scope, as strong aborts affect control flow at the beginning of a tick and not at the end of a tick. In this case, the trigger signal (say, \( S \)) is not tested when the delay node (\( N \)) is entered as the entering of \( N \) marks the end of a tick, and hence control would not even reach \( N \) if \( S \) was present.

However, \( S \) is tested when \( N \) is restarted at beginning of the next tick.

As already mentioned, we assume that the given program does not have cycles. However, what exactly constitutes a cycle in an Esterel program is not obvious, and to our knowledge there is no commonly accepted definition of cyclcity at the language level. The Esterel compilers that require acyclic programs differ in the programs they accept as “acyclic.” For example, the CEC accepts some programs that the v5 compiler rejects and vice versa [36], and a full discussion of this issue goes beyond the scope of this paper. Effectively, a program is considered cyclic if it is not (statically) schedulable—and compilers differ in their scheduling abilities. We here consider a program cyclic if the priority assignment algorithm presented in the next section fails. This results in the following definition, based on the CKAG.

**Definition 1 (Program cycle).** An Esterel program is cyclic if the corresponding CKAG contains a path from a node to itself, where for each node \( n \) and its successors along that path, \( n' \) and \( n'' \), the following holds:

\[
 n \in D \cup F \land n' \in n.\text{suc}_w \land \forall n \in F \land n' \in n.\text{suc}_c \cup n.\text{suc}_f \\
 \lor n \in T \land n' \in n.\text{suc}_c \cup n.\text{dep}_i \\
 \lor n \in T \land n' \in n.\text{dep}_d \land n'' \in n'.\text{suc}_c \cup n'.\text{suc}_f. 
\]  

Note that some of the sets that this definition uses are conservatively approximated by the compiler, as already mentioned in Section 3.3. In other words, our compiler may detect spurious cycles and therefore reject a program even if it is causal. As we consider dependencies only if they cross thread boundaries, it appears that we can schedule more programs than other compilers typically can, and we did not encounter a scheduling problem with any of the tested programs. However, a systematic investigation of this issue is still open.

4.2. Computing thread priorities

The task of the priority algorithm is to compute a priority assignment that respects the Esterel semantics as well as the execution model of the KEP. The algorithm computes for each reachable node \( n \) in the CKAG the priority \( n.\text{prio} \) and, for nodes in \( D \cup F \), \( n.\text{prionext} \). According to the Esterel semantics and the observations made in Section 3.3, a correct priority assignment must fulfill the following constraints, where \( m, n \) are arbitrary nodes in the CKAG.

**Constraint 1 (Dependencies).** A thread executing a dependency source node must have a higher priority than the corresponding sink. Hence, for \( m \in n.\text{dep}_i \), it must be \( n.\text{prio} > m.\text{prio} \), and for \( m \in n.\text{dep}_d \), it must be \( n.\text{prio} > m.\text{prionext} \).

**Constraint 2 (Intratrick priority).** Within a logical tick, a thread’s priority cannot increase. Hence, for \( n \in D \) and
After $\text{getPrio()}$ has been computed for all reachable nodes in the CKAG, a forall loop computes $\text{prionext}$ for reachable delay/fork nodes that have not been computed yet.

$(\text{via } \text{getPrio}())$ first checks whether it has already computed $n.\text{prio}$. If not, it then checks for a recursive call to itself (lines 3/4, see also Lemma 1). The remainder of $\text{getPrio}()$ computes $n.\text{prio}$ and, in case of delay and fork nodes, adds nodes to the $N_{\text{ToDo}}$ list. Similarly $\text{getPrioNext}()$ computes $n.\text{prionext}$.

**Lemma 1** (Termination). For a valid, acyclic Esterel program, $\text{getPrio}()$ and $\text{getPrioNext}()$ terminate. Furthermore, they do not generate a “Cycle detected!” error message.

**Proof (Sketch).** $\text{getPrio}()$ produces an error (line 4) if it has not computed $n.\text{prio}$ yet (checked in line 2) but has already been called (line 3) earlier in the call chain. This means that it has called itself via one of the calls to $\text{prioMax}()$ or $\text{prioNextMax}()$ (via $\text{getPrioNext}()$). An inspection of the calling pattern yields that an acyclic program in the sense of Definition 1 cannot yield a cycle in the recursive call chain. Since the number of nodes is finite, both algorithms terminate.

**Lemma 2** (Fulfillment of constraints). For a valid, acyclic Esterel program, the priority assignment algorithm computes an assignment that fulfills Constraints 1–4.

**Proof (Sketch).** First observe that—apart from the initialization in $\text{main}()$—each $n.\text{prio}$ is assigned only once. Hence, when $\text{prioMax}()$ returns the maximum of priorities for a given set of nodes, these priorities do not change any more. Therefore, the fulfillment of Constraint 1 can be...
deduced directly from getPrio(). Similarly for Constraint 2. Analogously getPrioNext() ensures that Constraints 3 and 4 are met.

Lemma 3 (Linearity). For a CKAG with \( N \) nodes and \( E \) edges, the computational complexity of the priority assignment algorithm is \( \Theta(N + E) \).

Proof (Sketch). Apart from the initialization phase, which has cost \( \Theta(N) \), the cost of the algorithm is dominated by the recursive calls to getPrio(). The total number of calls is bounded by \( E \). With an amortization argument, where the costs of each call are attributed to the callee, it is easy to see that the overall cost of the calls is \( \Theta(E) \).

Note also that while the size of the CKAG may be quadratic in the size of the corresponding Esterel program in the worst case, it is in practice (for a bounded abort nesting depth) linear in the size of the program, resulting in an algorithm complexity linear in the program size as well; see also the discussion in Section 6.2.

After priorities have been computed for each reachable node in the CKAG, we must generate code that ensures that each thread is executed with the computed priority. This task is relatively straightforward, Figure 7 shows the algorithm.

Another issue is the computation of thread ids, as these are also considered in scheduling decisions in case there are multiple threads of highest priority. This property is exploited by the scheduling scheme presented here, to avoid needless cycles. The compiler assigns increasing ids to threads during a depth-first traversal of the thread hierarchy; this is required in certain cases to ensure proper termination of concurrent threads [4].

4.3. Optimizations

Prior to running the priority/scheduling algorithm discussed before, the compiler tries to eliminate dependencies as much as possible. It does that using two mechanisms. The first is to try to be clever about the assignment of thread ids, as they are also used for scheduling decisions if there are multiple threads that have the highest priority (see Section 3.2). By considering dependencies between different threads, simple dependencies can be solved without any explicit priority changes. The second mechanism is to determine whether two nodes connected via a dependency are executable within the same instant. This is in general a difficult problem to analyze. We here only consider the special case where two nodes share some (least common) fork node, and one node has only instantaneous paths from that fork node, and the other node only not instantaneous paths. In this case, the dependency can be safely removed.

To preserve the signal-dependencies in the execution, additional priority assignments (PRIO statements) might have to be introduced by the compiler. To assure schedulability, the program is completely dismantled, that is, transformed into kernel statements. In this dismantled graph the priority assignments are inserted. A subsequent “undismantling” step before the computation of the WCRT detects specific patterns in the CKAG and collapses them to more complex instructions, such as AWAIT or SUSTAIN, which are also part of the KEP instruction set.

The KEP compiler performs a statement dismantling (see Section 3.1.2) as a preprocessing step. This facilitates code selection and also helps to eliminate spurious dependency cycles, and to hence increase the set of schedulable (accepted) programs, as already discussed in Section 4. After assigning priorities, the compiler tries again to “undismantle” compound statements whenever this is possible. This becomes apparent in the Edwards02 example; the AWAIT \( S\) \( T \) (Figure 3(c)) is the undismantled equivalent of the lines 7–9 in Edwards02-dism (Figure 3(b)).

The compiler suppresses PRIO statements for the main thread, because the main thread never runs concurrently to other threads. In the example, this avoids a PRIO 1 statement at label A3.

Furthermore, the compiler performs dead code elimination, also using the traversal results of the priority assignment algorithm. In the Edwards02 example, it determines that execution never reaches the infinite loop in line 32 of Edwards02-dism, because the second parallel thread never terminates normally, and therefore does not generate code for it.

However, there is still the potential for further optimizations, in particular regarding the priority assignment. In the Edwards02 program, one could for example hoist the PRIO \( 2\) \( T \) out of the enclosing loop, and avoid this PRIO statement altogether by just starting thread \( T \) with priority \( 2 \) and never changing it again. Even more effective would be to start \( T \) with priority 3, which would allow to undismantle L08–L12 into a single AWAIT.

5. WORST-CASE REACTION TIME ANALYSIS

Given a KEP program, we define its WCRT as the maximum number of KEP cycles executable in one instant. Thus WCRT analysis requires finding the longest instantaneous

Figure 7: Algorithm to annotate code with priority settings according to CKAG node priorities.
path in the CKAG, where the length metric is the number of required KEP instruction cycles. We abstract from signal relationships and might therefore consider unfeasible executions. Therefore, the computed WCRT can be pessimistic. We first present, in Section 5.1, a restricted form of the WCRT algorithm that does not handle concurrency yet. The general algorithm requires an analysis of instant reachability between fork and join nodes, which is discussed in Section 5.2, followed by the presentation of the general WCRT algorithm in Section 5.3.

### 5.1. Sequential WCRT algorithm

First we present a WCRT analysis of sequential CKAGs (no fork and join nodes). Consider again the ExSeq example in Figure 1(a).

The longest possible execution occurs when the signal l becomes present, as is the case in Tick 3 of the example trace shown in Figure 1(d). Since the abortion triggered by l is weak, the abort body is still executed in this instant, which takes four instructions: PAUSEL2, EMITL3, the GOTO4, and PAUSEL2 again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and EMITL5 and HALTL6 are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via a depth-first search (DFS) traversal of the CKAG, see the algorithm in Figure 8. For each node n a value n.inst is computed, which gives the WCRT from this node on in the same instant when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For noninstantaneous delay nodes, we distinguish two cases within a tick: control can reach a delay node d, meaning that the thread executing d has already executed some other instructions in that tick, or control can start in d, meaning that d must have been reached in some preceding tick. In the first case, the WCRT from d on within an instant is expressed by the d.inst variable already introduced. For the second case, an additional value d.next stores the WCRT from d on within an instant; "next" here expresses that in the traversal done to analyze the overall WCRT, the d.next value should not be included in the current tick, but in a next tick. Having these two values ensures that the algorithm terminates in the case of noninstantaneous loops: to compute d.next we might need the value d.inst.

For a delay node, we also have to take abortions into account. The handlers (i.e., their continuations—typically the end of an associated abort/trap scope) of weak abortions and exceptions are instantaneously reachable, so their WCRTs are added to the d.inst value. In contrast, the handlers of strong abortions cannot be executed in the same instant the delay node is reached, because according to the Esterel semantics an abortion body is not executed at all when the abortion takes place. On the KEP, when a strong abort takes place, the delay nodes where the control of the (still active) threads in the abortion body resides are executed once, and then control moves to the abortion handler. In other words, control cannot move from a delay node d to a (strong) abortion handler when control reaches d, but only when it starts in d. Therefore, the WCRT of the handler of a strong abortion is added to d.next, and not to d.inst.

We do not need to take a weak abortion into account for d.next, because it cannot contribute to a longest path.

---

**Figure 8:** WCRT algorithm, restricted to sequential programs. The nodes of a CKAG g are given by $N = T \cup L \cup D \cup F \cup J$ (see Section 3.3). g.root indicates the first KEP statement. cycles(stmt) returns the number of instruction cycles to execute stmt, see third column in Table 1.

```c
(1) int getWcrtSeq(g) // Compute WCRT for sequential CKAG g
   (2) for all n ∈ N do n.inst := n.next := ⊥ end
   (3) getInstSeq(g.root)
   (4) for all d ∈ D do getNextSeq(d) end
   (5) return max ((g.root.inst ∪ (d.next : d ∈ D))
   (6) end
```

```c
(1) int getInstSeq(n) // Compute statements instantaneously reachable from node n
   (2) if n.inst = ⊥ then
       (3) if n ∈ T ∪ L then
           (4) n.inst := max (getInstSeq(c) : c ∈ n.suc) + cycles(n.stmt)
       (5) elif n ∈ D then
           (6) n.inst := max (getInstSeq(c) : c ∈ n.suc_w ∪ n.suc_e) + cycles(n.stmt)
       (7) fi
       (8) fi
       (9) return n.inst
   (10) end
```

```c
(1) int getNextSeq(d) // from delay node d at tick start
   (2) if d.inst = ⊥ then
       (3) d.next := max (getInstSeq(c) : c ∈ d.suc_w ∪ d.suc_e) + cycles(d.stmt)
       (4) fi
       (5) return d.next
   (6) end
```
An abortion in an instant when a delay node is reached will always lead to a higher WCRT than an execution in a subsequent instant where a thread starts executing in the delay node.

The resulting WCRT for the whole program is computed as the maximum over all WCRTs of nodes where the execution may start. These are the start node and all delay nodes. To take into account that execution might start simultaneously in different concurrent threads, we also have to consider the next value of join nodes.

Consider again the example ExSeq in Figure 1. Each node \( n \) in the CKAG \( g \) is annotated with a label \( \langle W(n.\text{inst}) \rangle \) or, for a delay node, a label \( \langle W(n.\text{inst})/\langle n.\text{next} \rangle \rangle \). In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers \( L(n) \). It is \( g.\text{root} = L1 \). The sequential WCRT computation starts initializing the \( \text{init} \) and \( \text{next} \) values of all nodes to \( \bot \) (line 2 in \text{getWcrtSeq}, Figure 8). Then \text{getInstSeq}(L1) is called, which computes \( L1.\text{inst} := \text{max} \{ \text{getInstSeq}(L2) \} + \text{cycles}(\text{WABORTL1}) \). The call to \text{getInstSeq}(L2) computes and returns \( L2.\text{inst} := \text{cycles}(\text{PAUSEL2}) + \text{cycles}(\text{EMITL5}) + \text{cycles}(\text{HALTL4}) = 3 \), hence \( L1.\text{inst} := 3 + 2 = 5 \). Next, in line 4 of \text{getWcrtSeq}, we call \text{getNextSeq}(L2), which computes \( L2.\text{next} := \text{getInstSeq}(L3) + \text{cycles}(\text{PAUSEL2}) \). The call to \text{getInstSeq}(L3) computes and returns \( L3.\text{inst} := \text{cycles}(\text{EMITL3}) + \text{cycles}(\text{GOTOL4}) + L2.\text{inst} = 1 + 1 + 3 = 5 \). Hence \( L2.\text{next} := 5 + 1 = 6 \), which corresponds to the longest path triggered by the presence of signal \( l \), as we have seen earlier. The WCRT analysis therefore inserts an “\text{EMIT},\text{TICKLEN}, \#6” instruction before the body of the KEP assembler program to initialize the TickManager accordingly, as can be seen in Figure 1(c).

### 5.2. Instantaneous statement reachability for concurrent Esterel programs

It is important for the WCRT analysis whether a join and its corresponding fork can be executed within the same instant. The algorithm for instantaneous statement reachability computes for a source and a target node whether the target is reachable instantaneously from the source. Source and target have to be in sequence to each other, that is, not concurrent, to get correct results.

In simple cases like \text{EMIT} or \text{PAUSE} the sequential control flow successor is executed in the same instant respectively next instant, but in general the behavior is more complicated. The parallel, for example, will terminate instantaneously if all subthreads are instantaneous or an \text{EXIT} will be reached instantaneously; it is noninstantaneous if at least one subthread is not instantaneous.

The complete algorithm is presented in detail elsewhere [6]. The basic idea is to compute for each node three potential reachability properties: instantaneous, noninstantaneous, exit-instantaneous. Note that a node might be as well (potentially) instantaneous as (potentially) noninstantaneous, depending on the signal context. Computation begins by setting the instantaneous predicate of the source node to \( \text{true} \) and the properties of all other nodes to \( \text{false} \). When any property is changed, the new value is propagated to its successors. If we have set one of the properties to \( \text{true} \), we will not set it to \( \text{false} \) again. Hence the algorithm is monotonic and will terminate. Its complexity is determined by the amount of property changes which are bounded to three for all nodes, so the complexity is \( O(3*|N|) = O(|N|) \).

The most complicated computation is the property instantaneous of a join node, because several attributes have to be fulfilled for it to be instantaneous:

(i) For each thread, there has to be a (potentially) instantaneous path to the join node.

(ii) The predecessor of the join node must not be an \text{EXIT}, because \text{EXIT} nodes are no real control flow predecessors. At the Esterel level, an exception (\text{exit}) causes control to jump directly to the corresponding exception handler (at the end of the corresponding trap scope); this jump may also cross thread-boundaries, in which case all threads that contain the jump until the thread that contains the target of the jump and all their sibling threads terminate.

To reflect this at the KEP level, an \text{EXIT} instruction does not jump directly to the exception handler, but first executes the JOIN instructions on the way, to give them the opportunity to terminate threads correctly. If a JOIN is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler, or to another intermediate \text{JOIN}. To express this, we use the third property besides instantaneous and noninstantaneous: exit-instantaneous.

Roughly speaking, the instantaneous property is propagated via for-all quantifier, noninstantaneous and exit-instantaneous via existence-quantifier.

Most other nodes simply propagate their own properties to their successors. The delay node propagates in addition its noninstantaneous predicate to its delayed successors and exit nodes propagate exit-instantaneous reachability, when they themselves are reachable instantaneously.

### 5.3. General WCRT algorithm

The general algorithm, which can also handle concurrency, is shown in Figure 9. It emerges from the sequential algorithm that has been described in Section 5.1 by enhancing it with the ability to compute the WCRT of fork and join nodes. Note that the instantaneous of a join node is needed only by a fork node, all other transient nodes and delay nodes do not use this value for their WCRT. The WCRT of the join node has to be accounted for just once in the instantaneous WCRT of its corresponding fork node, which allows the use of a DFS-like algorithm.

The instantaneous WCRT of a fork node is simply the sum of the instantaneously reachable statements of its subthreads, plus the PAR statement for each subthread and the additional PARE statement.

The join nodes, like delay nodes, also have a next value. When a fork-join pair \((f, j)\) could be noninstantaneous, we have to compute a WCRT \( j.\text{next} \) for the next instant
Figure 9: General WCRT algorithm.

analogously to the delay nodes. Its computation requires first the computation of all subthread next WCRTs. Note that in case of nested concurrency these next WCRT values can again result from a join node. But at the innermost level of concurrency the next WCRT values all stem from delay nodes, which will be computed before the join next values. The delay next WCRT values are computed the same way as in the sequential case except that only successors within of the same thread are considered. We call successors of a different thread interthread-successors and their WCRT values are handled by the according join node. The join next value is the maximum of all interthread-successor WCRT values and the sum of the maximum next value for every thread.

If the parallel does not terminate instantaneously, all directly reachable states are reachable in the next instant. Therefore we have to add the execution time for all statements that are instantaneously reachable from the join node.

The whole algorithm computes first the next WCRT for all delay and join nodes; it computes recursively all needed next values. Thereafter the instantaneous WCRT for all remaining nodes is computed. The result is simply the maximum over all computed values.

Consider the example in Figure 2(a). First we note that the fork/join pair is always noninstantaneous, due to the PAUSEL6 statement. We compute $L6_{next} = \text{cycles}(\text{PAUSEL6}) + \text{cycles}(\text{EMITL7}) = 2$. From the fork node $L3$, the
PAR and PARE statements, the instantaneous parts of both threads and the JOIN are executed, hence \( L3.\text{inst} = 2 \times \text{cycles}(\text{PAR}) + \text{cycles}(\text{PARE}) + \text{cycles}(\text{JOIN}) + L4.\text{inst} + L5.\text{inst} = 2 + 1 + 1 + 1 + 2 = 7 \). It turns out that the WCRT of the program is \( L8.\text{next} = L6.\text{next} + L8.\text{inst} = 2 + 9 = 11 \). Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel programs is that due to the nesting of exceptions and concurrency, statements might be executed multiple times in one instant. This problem, also known as reincarnation, is handled correctly by our algorithm. Since we compute nested joins from inside to outside, the same statement may effect both the instantaneous and noninstantaneous WCRT, which are added up in the next join. This exactly matches the possible control-flow in case of reincarnation. Even when a statement is executed multiple times in an instant, we compute a correct upper bound for the WCRT.

Regarding the complexity of the algorithm, we observe that for each node its WCRT’s inst and next are computed at most once, and for all fork nodes a fork-join reachability analysis is additionally made, which has itself \( O(|N|) \). So we get altogether a complexity of \( O(|N| + |D| + |J|) + O(|F| \times |N|) = O(2 \times |N|) + O(|N|^2) = O(|N|^2) \).

### 5.4. Unreachable paths

Signal informations are not taken into account in the algorithms described above. This can lead to a conserva-
tive (too high) WCRT, because the analysis may consider unreachable paths that can never be executed. In Figure 10(a) we see an unreachable path increasing unnecessarily the WCRT because of demanding signal \( I \) present and absent instantaneously, which is inconsistent. Nevertheless there is no dead code in the graph, but only two possible paths regarding to path signal predicates.

Figure 10(b) shows an unreachable parallel path that leads to a too high WCRT of the fork node, because the sub-paths cannot be executed at the same time. Furthermore, the parallel is declared as possibly instantaneous, even though it is not. Therefore, all statements which are instantaneously reachable from the join node are also added.

Another unreachable parallel path is shown in Figure 10(c). This path is unreachable not because of signal informations but because of instantaneous behavior: the maximal paths of the two threads are never executed in the same instant. In other words, the system is never in a configuration (collection of states) such that both code segments become activated together. Instead of taking for each thread the maximum next WCRT and summing up, it would be more exact to sum up over all threads next WCRT’s executable instantaneously and then taking the maximum of these sums. Therefore we would have to enhance the reachability algorithm of the ability to determine how many ticks later a statement could be executed behind another. However, in this case the possible tick counts can become arbitrarily high for each node, so we would get a higher
complexity and a termination problem. Our analysis is conservative in simply assuming that all concurrent paths may occur in the same instant, and that all can be executed in the same instant as the join.

6. EXPERIMENTAL RESULTS

To evaluate the compilation and WCRT analysis approach presented here, we have implemented a compiler for the KEP based on the CEC infrastructure [34]. We will discuss in turn our validation approach and the quantitative results for the compiler, specifically the priority assignment scheme, and for the WCRT estimation.

6.1. Validation

To validate the correctness of the compilation scheme, as well as of the KEP itself, we have collected a fairly substantial validation suite, currently containing some 500 Esterel programs. These include all benchmarks made available to us, such as the Estbench [37], and other programs written to test specific situations and corner cases. An automated regression procedure compiles each program into KEP assembler, downloads it into the KEP, provides an input trace for the program, and records the output at each step. This output is compared to the results obtained from running the same program on a workstation, using Esterel Studio.

Table 2: Experimental results for the compiler and priority assignment. For each benchmark it lists the lines of code (LoC) for the source code, the lines of generated KEP assembler, the number of dependencies, the maximal nesting depth of abort scopes, the maximal degree of concurrency, the number of generated PRIO statements, the maximum priority of any thread, and the times for computing the priorities and for the overall compilation.

<table>
<thead>
<tr>
<th>Module name</th>
<th>LoC</th>
<th>Lines</th>
<th>Dependencies</th>
<th>Depth</th>
<th>KEP Max.Conc</th>
<th>#PRIO</th>
<th>Max.Prio</th>
<th>$t_{assign}$ [ms]</th>
<th>$t_{comp}$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd</td>
<td>152</td>
<td>167</td>
<td>36</td>
<td>2</td>
<td>4</td>
<td>30</td>
<td>3</td>
<td>14.9</td>
<td>2.7</td>
</tr>
<tr>
<td>abcddef</td>
<td>232</td>
<td>251</td>
<td>90</td>
<td>2</td>
<td>6</td>
<td>48</td>
<td>3</td>
<td>63.8</td>
<td>4.2</td>
</tr>
<tr>
<td>eight_buttons</td>
<td>332</td>
<td>335</td>
<td>168</td>
<td>2</td>
<td>8</td>
<td>66</td>
<td>3</td>
<td>72.3</td>
<td>5.9</td>
</tr>
<tr>
<td>channel_protocol</td>
<td>57</td>
<td>61</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>5.3</td>
<td>0.8</td>
</tr>
<tr>
<td>reactor_control</td>
<td>24</td>
<td>32</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>—</td>
<td>3.9</td>
<td>0.4</td>
</tr>
<tr>
<td>runner</td>
<td>26</td>
<td>38</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>—</td>
<td>4.4</td>
<td>0.4</td>
</tr>
<tr>
<td>wvw_button</td>
<td>94</td>
<td>134</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>10.0</td>
<td>1.6</td>
</tr>
<tr>
<td>tcint</td>
<td>410</td>
<td>472</td>
<td>65</td>
<td>5</td>
<td>17</td>
<td>45</td>
<td>3</td>
<td>112.2</td>
<td>17.3</td>
</tr>
</tbody>
</table>

Esterel Studio is also used to generate the input trace, using the “full transition coverage” mode. Note that the traces obtained this way still did not cover all possible paths. However, at this point we consider it very probable that a compilation approach that handles all transition coverage traces correctly would also handle the remaining paths. We also feel that this level of validation probably already exceeds the current state of the practice.

6.2. Compilation and priority assignment

As the emphasis here is more on the compilation approach and less on the underlying execution platform, we here refrain from a comparison of execution times and code sizes on the KEP versus traditional, nonreactive platforms; such a comparison can be found elsewhere [4]. Instead, we are here primarily interested in static code characteristics, and in particular how well the priority assignment algorithm works. Table 2 summarizes the experimental results for a selection of programs taken from the Estbench.

We note first that the generated code is very compact, and that the KEP assembler line count is comparable to the Esterel source. This is primarily a reflection on the KEP ISA, which provides instructions that directly implement most of the Esterel statements. Furthermore, the relationship between source code and KEP assembler size (and CKAG size) seems fairly linear. We note that the connection between program size and number of (interthread) dependencies is rather loose. For example, eight_buttons is smaller than tcint, but contains more than twice the number of dependencies. Next, we see that the maximal abort nesting depth tends to be limited, only in one case it exceeded three. The degree of concurrency again varied widely; not too surprisingly, the degree of concurrency also influenced the required number of PRIO statements (which—potentially—induce context switches). However, overall the number of generated PRIO statements seems acceptable compared to overall code size, and there were cases where we did not need PRIO at all, despite having several interthread dependencies. This reflects that the thread id assignment mechanism (see Section 4.3) is already fairly efficient in resolving dependencies. Similarly,
the assigned priorities tended to be low in general, for none of the benchmarks they exceeded three. Finally, the priority assignment algorithm and the overall compilation are quite fast, generally in the millisecond range.

6.3. Accuracy of WCRT analysis

As mentioned before, the WCRT analysis is implemented in the KEP compiler, and is used to automatically insert a correct EMIT,TICKLEN instruction at the beginning of the program, such that the reaction time is constant and as short as possible, without ever raising a timing violation by the TickManager. As discussed in Section 6.1, we measured the maximal reaction times and compared it to the computed value. Figure 11 provides a qualitative comparison of estimated and measured WCRT and measured ACRT, more details are given in Table 3. We have never underestimated the WCRT, and our results are on average 22% too high, which we consider fairly tight compared to other reported WCET results [22]. For each program, the lines of code, the computed WCRT and the measured WCRT with the resulting difference are given. We also give the average WCRT analysis time on a standard PC (AMD Athlon XP, 2.2 GHz, 512 KB Cache, 1 GB Main Memory); as the table indicates, the analysis takes only a couple of milliseconds.

The table also compares the ACRT with the WCRT. The ACRT is on average about two thirds of the WCRT, which is relatively high compared to traditional architectures. In other words, the worst case on the KEP is not much worse than the average case, and padding the tick length according to the WCRT does not waste too much resources. On the same token, designing for worst-case performance, as typically must be done for hard real-time systems, does not cause too much overhead compared to the typical average-case performance design. Finally, the table also lists the number of scenarios generated by Esterel-studio and accumulated logical tick count for the test traces.

7. CONCLUSIONS AND FURTHER WORK

We have presented a compiler for the KEP, and its integrated WCRT analysis. Since the KEP ISA is very similar to Esterel, the compilation of most constructs is straightforward. But the computation of priorities for concurrent threads is not trivial. The thread scheduling problem is related to the problem of generating statically scheduled code for sequential processors, for which Edwards has shown that finding efficient schedules is NP hard [9]. We encounter the same complexity, even though our performance metrics for an optimal schedule are a little different. The classical scheduling problem tries to minimize the number of context switches. On the KEP, context switches are free, because no state variables must be stored and resumed. However, to ensure that a program meets its dependency- implied scheduling constraints, threads must manage their priorities accordingly, and it is this priority switching which contributes to code size and costs an extra instruction at run time. Minimizing priority switches is related to classical constraint-based optimization problems as well as to compiler optimization problems such as loop invariant code motion.

We also have presented the WCRT analysis of Esterel programs. The restricted nature of Esterel and its sound mathematical semantics allow formal analysis of Esterel programs and make the computation of a WCRT for Esterel programs achievable. Our analysis is incorporated in the compiler and uses its internal graph representation, the concurrent KEP assembler graph (CKAG). In a first step we compute whether concurrent threads terminate instantaneously, thereafter we are able to compute for each statement how many instructions are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program. The analysis considers concurrency and the multiple forms of preemption that Esterel offers. The asymptotic complexity of the WCRT analysis algorithm is quadratic in the size of the program; however, experimental results indicate that the overhead of WCRT analysis as part of compilation is negligible. We have implemented this analysis into the KEP compiler, and use it to automatically compute an initialization value for the KEP’s TickManager. This allows to achieve a high- and constant-response frequency to the environment, and can also be used to detect hardware errors by detecting timing overruns.

Our analysis is safe, that is, conservative in that it never underestimates the WCRT, and it does not require any user annotations to the program. In our benchmarks, it overestimates the WCRT on average by about 22%. This is already competitive with the state of the art in general WCET analysis, and we expect this to be acceptable in most cases. However, there is still significant room for improvement. So far, we are not taking any signal status into account, therefore our analysis includes some unreachable paths. Considering all signals would lead to an exponential growth of the complexity, but some local knowledge should be enough to rule out most unreachable paths of this kind. Also a finer
grained analysis of which parts of parallel threads can be executed in the same instant could lead to better results. However, it is not obvious how to do this efficiently.

Our analysis is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. At least for nonparallel programs our approach should be of value for other compilation methods for Esterel as well, for example, simulation-based code generation. A virtual machine with similar support for concurrency could also benefit from our approach. We would also like to generalize our approach to handle different ways to implement concurrency. A WCRT analysis directly on the Esterel level gives information on the longest possible execution path. Together with a known translation to C, this WCRT information could be combined with a traditional WCET analysis, which takes caches and other hardware details into account.

To conclude, while we think that the approaches for compilation and WCRT analysis presented here are another step towards making reactive processing attractive, there are still numerous paths to be investigated here, including the application of these results towards classical software synthesis. A further issue, which we have not investigated here at all, is to formalize the semantics of reactive ISAs. This would help to deepen the understanding of reactive processing platforms, and could open the door towards new, interesting synchronous models of computation.

REFERENCES


Formal Analysis Tools for the Synchronous Aspect Language Larissa

David Stauch

Verimag, Centre Équation 2, Avenue de Vignate, 38610 Gières, France

Correspondence should be addressed to David Stauch, david.stauch@imag.fr

Received 17 September 2007; Accepted 18 April 2008

Recommended by Michael Mendler

We present two tools for the formal analysis of the aspect language Larissa, which extends the simple synchronous language Argos. The first tool concerns the combination of design-by-contract with Larissa aspects, and shows how we can apply an aspect not only to a program, but to a specification of programs in form of a contract, and obtain a new contract. The second concerns aspect interferences, that is, aspects that influence each other in an unintended way if they are applied to the same program. We present a way to weave aspects in a less conflict-prone manner, and a means to detect remaining conflicts statically. These tools are quite powerful, compared to those available for other aspect languages.

Copyright © 2008 David Stauch. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

Aspect-oriented programming (AOP) offers programming constructs to a base language, which aim at encapsulating crosscutting concerns. These are concerns that cannot be properly captured into a module by the decomposition offered by the base language. AOP languages express crosscutting concerns in aspects, and weave (i.e., compile) them in the base program with an aspect weaver. All the aspect extensions of existing languages (like AspectJ [1]) share two notions: pointcuts and advice. A pointcut describes, with a general property, the program points (called join points), where the aspect should intervene (e.g., all methods of the class X, or all methods whose names begin with set). The advice specifies what has to be done at each join point (e.g., execute a piece of code before the normal code of the method).

Reactive systems are control systems that are in constant interaction with their environment. They are often programmed in dedicated languages, which must fulfill specific requirements. First, reactive systems often fulfill safety-critical functions, and thus require the use of formal methods in their development. Programming languages for them must thus be formally defined, and have a connection to verification tools. Furthermore, they usually fulfill several tasks in parallel, and programming languages must thus offer an explicit parallel composition of components.

The family of synchronous languages are such dedicated languages, which are very successfully used to program safety-critical reactive systems, for example, control systems in airplanes or nuclear power stations. Synchronous languages are all based on the same semantic principle, the synchrony hypothesis, which divides time into instants and assumes that reactions of parallel components are atomic, that is, that outputs are emitted as soon as the inputs are received. A second principle is the synchronous broadcast, which allows outputs of a component to be read by other components in parallel. These principles allow to develop synchronous languages that are very expressive and have a clear and simple semantics with strong semantic properties. The family of synchronous languages contains languages with different styles, for example, the dataflow language Lustre [2] and the imperative language Esterel [3]. The simplest language of the family is Argos [4], a hierarchical automata language, based on Mealy machines, which can be composed by different operators.

There are also crosscutting concerns in synchronous programs, which cannot be encapsulated with the parallel composition and other operators of synchronous languages. They are, however, different from crosscutting concerns...
in programs written in general-purpose languages because they crosscut the parallel structure of reactive programs. Therefore, and because they are usually not formally defined, existing aspect languages cannot be applied to reactive systems. Thus we developed an aspect-oriented extension for Argos, called Larissa [5].

When designing Larissa, we took great care to give it a clean and simple semantics and strong semantic properties, as they are common in synchronous languages. Thus point-cuts are specified as synchronous observers [6], that is, Argos programs that, via the synchronous broadcast, observe the inputs and the outputs of the base program, and compute a safety property on them. This is a semantic and at the same time very expressive mechanism. Larissa has different kinds of advice, and all are specified depending only on the interface of the base program, but not on its implementation. Due to this semantic definition, Larissa aspects preserve the equivalence of base programs.

Having a clean and simple semantics has the advantage of making programs easier to understand for programmers. Furthermore, it allows the semantic analysis of programs. In this paper, we present two tools for semantic analysis.

The first combines Larissa aspects with another successful programming technique, design-by-contract [7], which has been originally introduced for object-oriented systems. There, a method is specified by a contract, which consists of an assumption clause and a guarantee clause. It fulfills its contract if after its execution, the guarantee holds whenever the assumption was true when the program was called.

Contracts have been adapted to reactive systems by [8], where assumptions and guarantees are expressed as observers, in the same way as Larissa pointcuts. Because reactive systems constantly receive inputs and emit outputs, it seems natural to let the assumption observer restrict the inputs, and let the guarantee observer ensure properties on the outputs.

Aspect-oriented programming and design-by-contract can hardly be used concurrently: when an aspect is applied to a method, it changes its semantics, such that its contract is no longer valid. The approach we present solves this problem for Argos and Larissa by generating a new contract that is valid after the application of the aspect. We show how to apply an aspect asp to a contract C, such that for any program P which fulfills C, P with asp fulfills C’. Although an observer is also an Argos program, we cannot directly apply aspects because it has a different interface where the outputs of the program have become inputs. We therefore transform the observers first into nondeterministic Argos programs, which produce exactly the execution traces the observer accepted, and apply the aspect to these. A second difficulty comes from the fact that we must treat assumption and guarantee differently to preserve the correctness of our algorithm. We demonstrate this approach on an example which models a tramway door controller.

The second semantic analysis we present treats interference between aspects. Applying several aspects to the same program may lead to unintended results because of conflicts between the aspects. We say that two aspects interfere when weaving them in different orders does not yield the same result.

Whether two aspects interfere depends on the way they are woven in the program. We distinguish sequential and joint weaving. Sequential weaving means weaving the aspects one by one, where the next aspect is woven in the result of the previous weaving. Argos operators are defined that way, and also Larissa aspects. On the other hand, joint weaving means weaving several aspects together, into the same base program. AspectJ is defined that way: its semantics is not defined as a transformation of the base program, but as injecting behavior in the running program, including other aspects.

Sequential weaving often causes interference between aspects because the second aspect is applied to the first, but not the other way round. Therefore, we present a joint weaving mechanism for Larissa, which applies aspects to the same base program, and thus reduces interferences. As opposed to AspectJ, however, all jointly woven aspects only affect the base program, but not each other. Therefore, we still need sequential weaving, in cases where one aspect needs to affect another.

Joint weaving removes many cases of interference, which we also demonstrate with an example. However, interference is unavoidable when two aspects want to modify the base program in the same point. Such cases should be made explicit to the programmer. We therefore present an interference analysis for jointly woven aspects, that can either determine that two aspects do not interfere for a given base program, or that they never interfere for any base program.

In the first case, we apply both pointcuts to the base program and check if there are common join points. In the second case, it is sufficient to perform a parallel product of the two pointcuts. All these steps must be performed during the compilation process anyway, and thus add no additional cost.

Both tools that we present in this paper are only possible because of the semantic definition of Larissa. Thus the contract weaving can apply aspects to programs whose implementation is unknown. The interference analysis also depends on the semantic definition of Larissa, notably on the precise description of join points with observers, which makes it possible to determine statically the points where several aspects want to introduce their advice.

The structure of the paper is as follows: Section 2 introduces Larissa and Argos, Section 3 shows how to weave contracts in aspects, and Section 4 contains an example for this. Next, Section 5 explains the interference analysis, using a second example. Section 6 discusses related work, and Section 7 concludes. Work on the combination of contracts and aspects has been published in [9], and work on aspect interference in [10].

2. ARGOS AND LARISSA

This section presents a restriction of the Argos language [4], and the Larissa extension [5]. Argos is defined as a set of operators on complete and deterministic input/output automata communicating via Boolean signals. The semantics
of an Argos program is given as a trace semantics that is common to a wide variety of reactive languages.

2.1. Traces and trace semantics

Definition 1 (Traces). Let \( I, \Theta \) be finite sets of Boolean input and output variables representing signals from and to the environment. An input trace, it, is a function: \( it : N \rightarrow \{ \text{true}, \text{false} \} \). An output trace, ot, is a function: \( ot : N \rightarrow \{ \text{true}, \text{false} \} \). One denotes by InputTraces (resp., OutputTraces) the set of all input (resp., output) traces. A pair \((it, ot)\) of input and output traces \((i/o\)-traces for short) provides the valuations of every input and output at each instant \( n \in N \). One denotes by \( i(n)[i] \) (resp., \( o(t)[n][o] \)) the value of the input \( i \in I \) (resp., the output \( o \in \Theta \)) at the instant \( n \in N \).

A set of pairs of \( i/o\)-traces \( S = \{(it, ot) \mid it \in \text{InputTraces} \land ot \in \text{OutputTraces}\} \) is deterministic if and only if for all \((it, ot), (it', ot') \in S\) \((it = it') \Rightarrow (ot = ot')\), and it is complete if and only if for all \( it \in \text{InputTraces} \) \( \exists ot \in \text{OutputTraces} \cdot (it, ot) \in S\).

A set of traces is a way to define the semantics of an Argos program \( P \), given its inputs and outputs. From the above definitions, a program \( P \) is deterministic if from the same sequence of inputs it always computes the same sequence of outputs. It is complete whenever it allows every sequence of every eligible valuations of inputs to be computed. Determinism is related to the fact that the program is indeed written with a programming language (which has deterministic execution); completeness is an intrinsic property of the program that has to react forever, to every possible inputs without any blocking.

2.2. Argos

The core of Argos is made of input/output automata, the synchronous product, and the encapsulation. The synchronous product allows to put automata in parallel which synchronize on their common inputs, and the encapsulation is the operator that expresses the communication between automata. The semantics of an automaton is defined by a set of traces, and the semantics of the operators is given by translating expressions into flat automata.

Definition 2 (Automaton). An automaton \( \mathcal{A} \) is a tuple \( \mathcal{A} = (Q, s_{\text{init}}, I, \Theta, T) \), where \( Q \) is the set of states, \( s_{\text{init}} \in Q \) is the initial state, \( I \) and \( \Theta \) are the sets of Boolean input and output variables, respectively, \( T \subseteq Q \times \text{Boo} \times T \times Q \) is the set of transitions. \( \text{Boo}(I) \) denotes the set of Boolean formulas with variables in \( I \). For \( t = (s, \ell, O, s') \in T \), \( s, s' \in Q \) are the source and target states, \( \ell \in \text{Boo}(I) \) is the triggering condition of the transition, and \( O \subseteq \Theta \) is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.

The semantics of an automaton \( \mathcal{A} = (Q, s_{\text{init}}, I, \Theta, T) \) is given in terms of a set of pairs of \( i/o\)-traces. This set is built using the following functions:

\[
\begin{align*}
S_{\text{step}} &: Q \times \text{InputTraces} \times N \rightarrow 2^Q, \\
O_{\text{step}} &: Q \times \text{InputTraces} \times N \setminus \{0\} \rightarrow 2^\Theta.
\end{align*}
\]

where \( S_{\text{step}}(s, it, n) \) returns the set of states that are reachable from state \( s \) after performing \( n \) steps with the input trace \( it \); \( O_{\text{step}}(s, it, n) \) contains the different combinations of outputs that can be emitted at step \( n \) after executing it from \( s \):

\[
\begin{align*}
n = 0 &: S_{\text{step}}(s, it, n) = \{s\}, \\
0 < n &: S_{\text{step}}(s, it, n), O \subseteq O_{\text{step}}(s, it, n), \\
&\quad \land \exists s' \in S_{\text{step}}(s, it, n-1), \\
&\quad \land \ell \in \text{Boo}(I), O, s' \in T, \\
&\quad \land \ell \in \Theta
\end{align*}
\]

Note that if the automaton is deterministic and complete, \( S_{\text{step}} \) and \( O_{\text{step}} \) always return a set with a single element.

We denote Traces(\( \mathcal{A} \)) the set of all traces built following this scheme: Traces(\( \mathcal{A} \)) defines the semantics of \( \mathcal{A} \). The automaton \( \mathcal{A} \) is said to be deterministic (resp., complete) if and only if its set of traces Traces(\( \mathcal{A} \)) is deterministic (resp., complete) (see Definition 1). Two automata \( \mathcal{A}_1, \mathcal{A}_2 \) are trace-equivalent, noted \( \mathcal{A}_1 \sim \mathcal{A}_2 \), if and only if Traces(\( \mathcal{A}_1 \)) = Traces(\( \mathcal{A}_2 \)). We assume that Argos programs are deterministic and complete, as these are important properties for reactive systems.

Definition 3 (synchronous product). Let \( \mathcal{A}_1 = (Q_1, s_{\text{init}_1}, I_1, \Theta_1, T_1) \) and \( \mathcal{A}_2 = (Q_2, s_{\text{init}_2}, I_2, \Theta_2, T_2) \) be automata. The synchronous product of \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) is the automaton \( \mathcal{A}_1 \parallel \mathcal{A}_2 = (Q_1 \times Q_2, (s_{\text{init}_1}, s_{\text{init}_2}), I_1 \cup I_2, \Theta_1 \cup \Theta_2, T) \), where \( T \) is defined by

\[
(s_1, s_2, O_1, s'_1) \in T_1 \land (s_2, s_2, O_2, s'_2) \in T_2 \Leftrightarrow ((s_1, s_2), s_1 \land O_1 \lor O_2, (s'_1, s'_2)) \in T.
\]

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and completeness.

Definition 4 (Encapsulation). Let \( \mathcal{A} = (Q, s_{\text{init}}, I, \Theta, T) \) be an automaton and \( \Gamma \subseteq I \cup \Theta \) be a set of inputs and outputs of \( \mathcal{A} \). The encapsulation of \( \mathcal{A} \) w.r.t. \( \Gamma \) is the automaton \( \mathcal{A} \backslash \Gamma = (Q, s_{\text{init}}, I \setminus \Gamma, \Theta \setminus \Gamma, T') \), where \( T' \) is defined by

\[
(s, \ell, O, s') \in T \land \ell^+ \cap \Gamma \subseteq O \land \ell^- \cap \Gamma \cup O = \varnothing \Leftrightarrow (s, \exists \Gamma \cdot \ell, O \setminus \Gamma, s') \in T',
\]

where \( \ell^+ \) is the set of variables that appear as positive elements in the monomial \( \ell \) (i.e., \( \ell^+ = \{ x \in I \mid (x \land \ell) = \ell \} \)). \( \ell^- \) is the set of variables that appear as negative elements in the monomial \( I \) (i.e., \( \ell^- = \{ x \in I \mid (x \land \ell) = \ell \} \)). \( \exists \Gamma \cdot \ell \) is then defined as \( \exists \Gamma \cdot \ell = \land_{\ell \in \ell^+} \land \land_{\ell \in \ell^-} \ell \).

Intuitively, a transition \( (s, \ell, O, s') \in T \) is still present in the result of the encapsulation operation if its label satisfies a
local criterion made of two parts: \( \ell^+ \cap \Gamma \subseteq O \) means that a local variable which needs to be true has to be emitted by the same transition; \( \ell^- \cap \Gamma \cap O = \emptyset \) means that a local variable that needs to be false should not be emitted in the transition. If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by \( \exists ! \Gamma \cdot \ell^+ \) for the input part, and by \( \ell^- \cap \Gamma \cap O \) for the output part. In general, the encapsulation operation does not preserve determinism nor completeness. This is related to the so-called “causality” problem intrinsic to synchronous languages (see, e.g., [3]).

2.2.1. An example

Figure 1(a) shows a 3-bits counter. Dashed lines denote parallel compositions and the overall box denotes the encapsulation of the three parallel components, hiding signals \( b \) and \( c \). The idea is the following: the first component on the right receives \( a \) from the environment, and sends \( b \) to the second one, every two \( a \)'s. Similarly, the second one sends \( c \) to the third one, every two \( b \)'s. \( b \) and \( c \) are the carry signals. The global system has \( a \) as input and \( d \) as output; it counts \( a \)'s modulo 8, and emits \( d \) every 8 \( a \)'s. Applying the semantics of the operator (first the product of the three automata, then the encapsulation) yields the simple flat automaton with 8 states (Figure 1(b)).

2.3. Larissa

Argos operators are already powerful. However, there are cases in which they are not sufficient to modularize all concerns of a program: a small modification of the global program’s behavior may require that we modify all parallel components, in a way that is not expressible with the existing operators.

The goal of aspects being to specify such cross-cutting modifications, we proposed an aspect-oriented extension for Argos [5], which allows the modularization of a number of recurrent problems in reactive programs, like the reinitialization. This leads to the definition of a new operator (the aspect weaving operator), which preserves determinism and completeness of programs, as well as semantic equivalence between programs. Similar to aspects in other languages, a Larissa aspect consists of a pointcut, which selects a set of join points, and an advice, which modifies these join points.

2.3.1. Join point selection

To preserve semantical equivalence, pointcuts in Larissa are not expressed in terms of the internal structure of the base program (as, e.g., state names), but refer to the observable behavior of the program only, that is, its inputs and outputs. In the family of synchronous languages, where the communication between parallel components is the synchronous broadcast, observers [6] are a powerful and well-understood mechanism which may be used to describe pointcuts. Indeed, an observer is a program that may observe the inputs and the outputs of the base program, without modifying its behavior, and computes a safety property (in the sense of safety/liveness properties as defined in [11]).

Therefore, observers are well suited to express pointcuts. A pointcut is thus an observer which selects a set of join point transitions by emitting a single output \( JP \), the join point signal. A transition \( T \) in a program \( P \) is selected as a join point transition when in the concurrent execution of \( P \) and the pointcut, \( JP \) is emitted when \( T \) is taken.

Technically, we perform a synchronous product between the program and the pointcut and select those transitions in the product which emit \( JP \). However, if we simply put a program \( P \) and an observer \( PC \) in parallel, \( P \)'s outputs \( O \) will become synchronization signals between them, as they are also inputs of \( PC \). They will be encapsulated, and are thus no longer emitted by the product. We avoid this problem by introducing a new output \( o' \) for each output \( o \) of \( P : o' \) will be used for the synchronization with \( PC \), and \( o \) will still be visible as an output. First, we transform \( P \) into \( P' \) and \( PC \) into \( PC' \), where for all \( o \in O \), \( o \) is replaced by \( o' \). Second, we duplicate each output of \( P \) by putting \( P \) in parallel with one single-state automaton per output \( o \) defined by \( \text{dupl}_o = (\{q\},\{\{o'\}\},\{o\},\{o',o,q\}) \). The complete product, where \( O \) is noted \( \{o_1,\ldots,o_n\} \), is given by

\[
\mathcal{P}(P,PC) = (P'\|PC'||\text{dupl}_{o_1}\|\cdots\|\text{dupl}_{o_n}) \\setminus \{o'_1,\ldots,o'_n\}.
\]

(5)

The join point transitions are those transitions of \( \mathcal{P}(P,PC) \) that emit \( JP \). Figure 2 illustrates the pointcut mechanism. The pointcut (b) specifies any transition which emits \( c \) in base program (a), the loop transition in state \( B \) is selected as a join point transition.

2.3.2. Specifying the advice

In aspect-oriented languages, the advice expresses the modification applied to the base program. In Larissa, we define

![Figure 1](image1.png)

**Figure 1**: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions for example, the transition labelled by “a/b” is triggered when a is true and emits b.

![Figure 2](image2.png)

**Figure 2**: Example pointcut.
two types of advice: in the first type, an advice replaces the join point transitions with advice transitions pointing to an existing target state; in the second type, an advice introduces an Argos program between the source state of the join point transition and an existing target state. In both cases, target states have to be specified without referring explicitly to state names.

An advice adv has two ways of specifying the target state \( T \) among the existing states of the base program \( P \). \( T \) is the state of \( P \) that would be reached by executing a finite input trace from either the initial state of \( P \), adv is then called toInit advice, or from the source state of the join point transition, adv is then called toCurrent advice. As the base program is deterministic and complete, executing an input trace from any of its states defines exactly one state.

The advice weaving operator \( \prec_{JP} \) adv weaves a piece of advice adv in a program.

**Advice transitions**

The first type of advice consists in replacing each join point transition with an advice transition. Once the target state is specified by a finite input trace \( \sigma = \sigma_1 \cdots \sigma_n \), the only missing information is the label of these new transitions. We do not change the input part of the label, so as to keep the woven automaton deterministic and complete, but we replace the output part by some advice outputs \( O_{adv} \). These are the same for every advice transition, and are thus specified in the aspect. Advice transitions are illustrated in Figure 3.

**Formal definition**

We only define toInit advice formally. A formal definition of the complete Larissa language can be found in [12, Chapter 4].

**Definition 5** (toInit advice weaving). Let \( \mathcal{A} = (Q, s_{init}, I, \varnothing, \mathcal{T}) \) be a deterministic and complete automaton and adv = \( (O_{adv}, \text{toInit}, \sigma) \) a piece of advice, with \( \sigma : [0, \ldots, \ell_{\sigma}] \rightarrow \{\text{true}, \text{false}\} \) a finite input trace of length \( \ell_{\sigma}+1 \). The advice weaving operator \( \prec_{JP} \) adv on \( \mathcal{A} \) and returns the automaton \( \prec_{JP} \) adv = \( (Q, s_{init}, I, \varnothing \cup O_{adv}, \mathcal{T}') \), where \( \mathcal{T}' \) is defined as follows, with \( \{\text{targ}\} = S_{\text{step}} A (s_{init}, \sigma, \ell_{\sigma}) \) being the new target state:

\[
\begin{align*}
(s, \ell, O, s') \in T \wedge JP \notin O & \Rightarrow (s, \ell, O, s') \in T', \quad (6) \\
(s, \ell, O, s') \in T \wedge JP \in O & \Rightarrow (s, \ell, O_{adv}, \text{targ}) \in T'. \quad (7)
\end{align*}
\]

Transitions (6) are not join point transitions and are left unchanged. Transitions (7) are the join point transitions, their final state \( \text{targ} \) is specified by the finite input trace \( \sigma \). \( S_{\text{step}} A \) (which has been naturally extended to finite input traces) executes the trace during \( \ell_{\sigma} \) steps, from the initial state of \( A \).

**Advice programs**

It is sometimes not sufficient to modify single transitions, that is, to jump to another location in the automaton in only one step. It may be necessary to execute arbitrary code when an aspect is activated. In these cases, we can insert an automaton between the join point and the target state.

Therefore, we use an inserted automaton \( A_{ins} \) that can terminate. Since Argos has no built-in notion of termination, the programmer of the aspect has to identify a final state \( F \) (denoted by filled black circles in the figures).

We first specify a target state \( T \) as explained above. Then, for every \( T \), a copy of the automaton \( A_{ins} \) is inserted, which means: (1) replace every join point transition \( J \) with target state \( T \) by a transition to the initial state \( I \) of this instance of \( A_{ins} \). As for advice transitions, the input part of the label is unchanged and the output part is replaced by \( O_{adv} \); (2) connect the transitions that went to the final state \( F \) in \( A_{ins} \) to \( T \). Advice programs are illustrated in Figure 4.

2.3.3. Fully specifying an aspect

An aspect is given by the specification of its pointcut and its advice: \( \text{asp} = (PC, \text{adv}) \), where PC is the pointcut and adv is the advice. adv is a tuple which contains: (1) the advice outputs \( O_{adv} \); (2) the type of the target state specification (toInit or toCurrent); (3) the finite trace \( \sigma \) over the inputs of the program; and optionally, (4) \( P_{adv} \), the advice program. Thus advice can be a tuple \( (O_{adv}, \text{type}, \sigma) \), or, with an advice program, a tuple \( (O_{adv}, \text{type}, \sigma, P_{adv}) \), with type \( \in \{\text{toCurrent, toInit}\} \). An aspect is woven into a program by
first determining the join point transitions and then weaving the advice.

Definition 6 (Aspect weaving). Let \( P \) be a program and \( \text{asp} = (\text{PC}, \text{adv}) \) an aspect for \( P \). The weaving of \( \text{asp} \) on \( P \) is defined by

\[
P \bowtie \text{asp} = \mathcal{P}(P, \text{PC}) \bowtie_{\text{JP}} \text{adv}. \tag{8}\]

2.3.4. Example

As an example, consider a monostable flip-flop (MFF) with one input \( a \) and one output \( b \), which emits two \( bs \) after it received an \( a \). Figure 5(a) shows an implementation of the MFF in Argos. We want to make the MFF retriggerable, meaning that if an \( a \) is emitted during several following instants, the MFF continues emitting \( b \). We do this by applying the aspect \( A_{\text{tri}} = (\text{PC}, (b, \text{toInit}, (a))) \) to the MFF, where \( \text{PC} = ([S], S, \{a, b\}, \{\text{JP}\}, ([S, a-b, \text{JP}, S]) \) is a pointcut which selects all occurrences of \( a-b \) as join points. Figure 5(b) shows the result of applying \( A_{\text{tri}} \) to the implementation.

3. COMBINING CONTRACTS AND ASPECTS

In this section, we show how to apply aspects to a specification of programs in form of a contract. First, we formally define contracts for Argos, then explain informally how to weave aspects into them, and finally define this process formally.

3.1. Contracts for Argos

The observers we use in contracts are slightly different from those used as pointcuts. Notably, once they start emitting their output \( \text{err} \), they continue emitting it forever. This is done in an error state error. Such an observer specifies a class of programs fulfilling a certain safety property, namely, those programs where the observer never emits \( \text{err} \) when combined with them. The error state is thus a way to refuse certain inputs while keeping the observer complete.

Definition 7 (observer). An observer is an automaton \((Q \cup \{\text{Error}\}, q_0, I \cup \Theta, \{\text{err}\}, T)\) which observes an automaton with inputs \( I \) and outputs \( \Theta \). When an observer emits \( \text{err} \), it will go to state error and also emit \( \text{err} \) in the next instant. A program \( P \) is said to obey an observer \( \text{obs} \) (noted \( P \models \text{obs} \)) if and only if \( P|\text{obs} \setminus \Theta \) produces no trace which emits \( \text{err} \).

Transitions leading to the error state are called error transitions. A contract specifies a class of programs with two observers, an assumption and a guarantee. Definition 8 is an auxiliary definition, used to formally define contracts in Definition 9. \( \diamond \) denotes the trace for a single output \( \text{err} \) that never emits \( \text{err} \), that is, \( \diamond_{\text{err}}[n] = \text{false} \) for all \( n \). An observer that accepts a trace emits \( \diamond \).

Definition 8 (trace combination). Let it : \( N \rightarrow I \rightarrow \{\text{true}, \text{false}\} \) and ot : \( N \rightarrow (I \cup \Theta) \rightarrow \{\text{true}, \text{false}\} \) be traces, with \( I \cap \Theta = \emptyset \). Then, it-ot : \( N \rightarrow (I \cup \Theta) \rightarrow \{\text{true}, \text{false}\} \) is a trace s.t. for all \( i \in I \), it-ot(n)(i) = it(n)(i) \& for all \( o \in \Theta \), it-ot(n)(o) = \text{false} for all \( o \in \Theta \).

Definition 9 (contract). A contract over inputs \( I \) and outputs \( \Theta \) is a tuple \((A, G)\) of two observers over \( I \cup \Theta \), where \( A \) is the assumption and \( G \) is the guarantee. A program \( P \) fulfills a contract \((A, G)\), written \( P \models (A, G) \), if and only if

\[
(\textit{it-ot, } \diamond) \in \text{Traces}(A) \land (\textit{it, ot}) \in \text{Traces}(P) \land \text{it-ot, } \diamond
\in \text{Traces}(G).
\]

Intuitively, a guarantee \( G \) should only restrict the outputs of a program and an assumption \( A \) should only restrict the inputs. We do not require this formally, but contracts which do not respect this constraint are of little use. Indeed, if \( G \) restricts the inputs more than \( A \), it follows from Definition 9 that there exists no program \( P \) s.t. \( P \models (A, G) \). Conversely, a program is usually placed in an environment \( E \), s.t. \( E \models A \). If \( A \) restricts the outputs, no such \( E \) exists, as the outputs are controlled by \( P \).

As an example for a contract, consider the following contract for the MFF from Section 2.3.4. The contract is composed of an assumption, shown in Figure 6(a), which states that \( a \)’s always occur in pairs, and a guarantee consisting of two automata, shown in Figures 6(b) and 6(c), which are composed in parallel. The automaton in Figure 6(b) guarantees that a single \( b \) is never emitted, and the automaton in Figure 6(c) guarantees that when \( a \) occurs while no \( b \) is emitted, \( b \) is emitted in the next instant. The product of Figures 6(b) and 6(c) is shown in Figure 6(d).

3.2. Weaving aspects in contracts

We want to apply an aspect \( \text{asp} \) not to a specific program \( P \), but to a class of programs defined by a contract \( C \), and obtain...
a new class of programs, defined by a contract $C'$, such that $P \models C \Rightarrow P \models C'$. To construct $C'$, we simulate the effect that the aspect has on a program as far as possible on the assumption and the guarantee observers of $C$. However, an aspect cannot be applied directly to an observer because the aspect has been written for a program with inputs $I$ and outputs $\Theta$, whereas for the observer, $\Theta$ are also inputs.

Therefore, we transform the observers of the contract first into nondeterministic automata (NDA), which produce exactly those traces that the observer accepts. We then weave the aspects into the NDA, with a modified definition of the weaving operator. The woven NDA are then transformed back into observers. The obtained observers may still be nondeterministic, and are thus determined.

Except for aspect weaving, all of these steps are different for the assumption and the guarantee, as far as the Error transitions are concerned. This is because the assumption and the guarantee have different functions in a contract: the assumption states which part of the program is defined by the contract, and the guarantee gives properties that are always true for this part. Indeed, a contract $(A, G)$ can be rewritten as $(\text{true}, A \Rightarrow G)$, where $A \Rightarrow G$ is an observer that emits $\text{err}$ when $G$ emits $\text{err}$ but not $A$. Thus the assumption can be considered as a negated guarantee.

After weaving an aspect, the assumption must exclude the undefined part of any program which fulfills the contract. Therefore, it must reject a trace (by emitting $\text{err}$) as soon as there exists a program for which it cannot predict the behavior. The guarantee, on the other hand, emits $\text{err}$ only for traces which cannot be emitted by any program which fulfills the contract. Therefore, after weaving an aspect, the new guarantee may only emit $\text{err}$ if it is sure that there exists no program that produces the trace.

On the other hand, we want the assumption to be as permissive as possible, to include all possible programs, and the guarantee as restrictive as possible, to characterize the woven program as exactly as possible. Thus when we know exactly the behavior of the program, as, for example, that of an inserted advice program, we do not emit $\text{err}$ in the assumption, but we emit $\text{err}$ in the guarantee to exclude all input/output combinations that are never produced by the program.

### 3.3. Formal definitions

This section describes the weaving of aspects into contracts in detail, and illustrates it on the MFF example. First, Definition 10 defines the transformation of an observer into an NDA through two functions, one for guarantee observers and one for assumption observers.

**Definition 10** (observer to NDA transformation). Let $\text{obs} = (Q \cup \{\text{Error}\}, q_0, I \cup \Theta, \{\text{err}\}, T)$ be an observer with an error state $\text{Error}$ over inputs $I$ and outputs $\Theta$, with $I \cap \Theta = \emptyset$. $\text{ND}_G(\text{obs}) = (Q, q_0, I, \Theta, T_{ND_G})$ defines an NDA, where $T_{ND_G}$ is defined by $(s, \ell_t \land \ell_o, o, s') \in T \Rightarrow (s, \ell_t, \ell_o, s') \in T_{ND_G}$. $\text{ND}_A(\text{obs}) = (Q \cup \{\text{Error}\}, q_0, I, T_{ND_A})$ defines an NDA, where $T_{ND_A}$ is defined by $(s, \ell_t \land \ell_o, o, s') \in T \Rightarrow (s, \ell_t, \ell_o, o, s') \in T_{ND_A}$.

Note that the transitions in obs which emit $\text{err}$ (i.e., the error transitions) have no corresponding transitions in $\text{ND}_G(\text{obs})$. In the guarantee, these transitions correspond to input/output combinations which are never produced by the program and must not be considered by the aspect. The other transitions are transformed such that part of the condition concerning $\Theta$ disappears, and those outputs that appeared as positive atoms in the condition (i.e., $\ell_o^D$) become outputs.

As an example, consider the guarantee of the MFF (Figure 6(d)). Its transformation into an NDA is shown in Figure 7(a). Note that the error state and the transitions leading to it have disappeared, and that $b$ is now an output. Thus the transition label $b$ has been transformed to $\text{true}/b$, and label $a\cdot\text{err}$ to $a$.

In the assumption, on the other hand, the error transition corresponds to inputs from the environment to which the program may react arbitrarily. If the aspect replaces these transitions in the assumption, they are also replaced in the program, and can thus be accepted from the environment by the woven program. Thus error transitions are not removed in $\text{ND}_A(\text{obs})$, so that the aspect weaving can modify them. The transformation of the assumption of the MFF (Figure 6(a)) is shown in Figure 8(a).

We can now apply an aspect to an NDA. However, a trace may lead to several states. Thus for each join point transition, several advice transitions must be created, one for each target state. We only give a definition for toInit advice, but the extension to toCurrent advice and advice programs is straightforward, and can be found in [12, Chapter 8].

**Definition 11** (toInit weaving for NDA). Let $A = (Q, s_{\text{init}}, I, \Theta, T)$ be an automaton and $\text{adv} = (O_{\text{adv}}, \text{toInit}, \sigma)$ a piece
of tolnt advice, with \( \sigma : [0, \ldots, \ell_\sigma - 1] \rightarrow [\text{true}, \text{false}] \) a finite input trace of length \( \ell_\sigma + 1 \). The advice weaving operator \( \triangleright \), weaves adv into \( A \) and returns the automaton \( A \triangleright \text{adv} = (Q, s_{\text{init}}, I, \emptyset \cup O_{\text{adv}}, T') \), where \( T' \) is defined as follows:

\[
(s, \ell, O, s') \in T \land J P \perp O \quad \text{and} \quad (s, \ell, O, s') \in T', \tag{10}
\]

\[
((s, \ell, O, s') \in T \land J P \in O) \forall \varrho \in S_{\text{step}_A}(s_{\text{init}}, \sigma, \ell_\sigma) \cdot (s, \ell, O_{\text{adv}}, \text{targ}) \in T'. \tag{11}
\]

Transitions (10) are not join point transitions and are left unchanged. Transitions (11) are the join point transitions, their final state tag is specified by the finite input trace \( \sigma \). \( S_{\text{step}_A} \) (which has been naturally extended to finite input traces) executes the trace during \( \ell_\sigma \) steps, from the initial state of \( A \). Figures 7(b) and 8(b) show the NDAs from our example with the retriggerable aspect from Section 2.3.4 woven into them. For both NDAs, the trace leads to a single point transition.

When transforming an NDA to an assumption, we do not add additional error transitions, but only leave those already there. In the guarantee, we add transitions to the error state from every state where the automaton is not complete. This is correct, as these transitions correspond to traces that are never produced by any program.

**Definition 12** (NDA to guarantee transformation). Let \( \text{nd} = (Q, q_0, I, \emptyset, T) \) be an NDA. \( \text{OBS}_G(\text{nd}) = (Q \cup \{\text{Error}\}, q_0, I \cup \emptyset \cup \{\text{err}\}, T' \cup T'') \) defines an observer, where \( T' \) and \( T'' \) are defined by

\[
(s, \ell, o, s') \in T(s, \ell \land \ell_{\sigma} \land \ell_{\overline{\sigma}}(s, o, s') \in T', \tag{12}
\]

\[
(s, \ell, \emptyset, s') \notin T' \land s \in Q \land \ell \text{ is a complete monomial over } I \cup \emptyset \land (s, \ell, \{\text{err}\}, \text{Error}) \in T'',
\]

where \( l_o = \bigwedge_{o \in O} o \) and \( l_{\overline{\sigma}} = \bigwedge_{o \in O} \overline{o} \) for a set \( O \) of variables.

**Definition 13** (NDA to assumption transformation). Let \( \text{nd} = (Q, q_0, I, \emptyset \cup \{\text{err}\}, T) \) be an NDA. \( \text{OBS}_A(\text{nd}) = (Q, q_0, I \cup \emptyset \cup \{\text{err}\}, T') \) defines an observer, where \( T' \) is defined by

\[
(s, \ell, o \cup e, s') \in T \land o \in \emptyset \land e \in \{\text{err}\} \quad \text{and} \quad (s, \ell, e \land \ell_{\sigma}(s, o, s') \in T'. \tag{13}
\]

Figures 7(c) and 8(c) show the NDAs from our example transformed back into observers. As expected, the obtained guarantee in Figure 7(c) tells us that whenever the program receives an \( a \), it emits \( b \)’s the two following instants. The assumption, however, requires that if an \( a \) is emitted, it continues to be emitted until there is no \( b \).

The resulting observer may not be deterministic. However, it can be made deterministic, as observers are acceptor automata. Determinization for guarantees and assumptions.
is different: a guarantee must only emit err for a trace σ if all programs fulfilling the contract never emit σ, and an assumption must emit err if there exists a program fulfilling the contract which is not defined for σ.

Existing determinization algorithms can be easily adapted to fulfill these requirements. We do not detail such algorithms here, but instead give conditions the determinization for assumptions and guarantees must fulfill. The new assumption and the new guarantee in the example are already deterministic, thus there is no need to determinize them.

The assumption determinization gives precedence to error transition. If there is a choice between an error transition and a nonerror transition, the error transition is always taken. Thus the determinized assumption only accepts a program if all possible nondeterministic executions of the nondeterminized assumption accept it.

Definition 14 (assumption determinization). Let M be an NDA with outputs \{err\}. Det_A(M) is a deterministic automaton such that

\[
\begin{align*}
\text{(it, ot)} & \in \text{Traces}(\text{Det}_A(M)) \iff \\
\text{(it, ot)} & \in \text{Traces}(M) \land \exists \text{ot}' \cdot (\text{it, ot}') \\
& \in \text{Traces}(M) \land \text{ot}'(n)[\text{err}] \\
& = \text{true} \land \text{ot}(n)[\text{err}] = \text{false}.
\end{align*}
\]

(14)

As opposed to the assumption determinization, the guarantee determinization gives precedence to nonerror transitions over error transitions. Thus the determinized guarantee emits err only if all possible executions of the nondeterminized guarantee also emit err.

Definition 15 (guarantee determinization). Let M be an NDA with outputs \{err\}. Det_G(M) is a deterministic automaton such that

\[
\begin{align*}
\text{(it, ot)} & \in \text{Traces}(\text{Det}_G(M)) \iff \\
\text{(it, ot)} & \in \text{Traces}(M) \land \exists \text{it}' \cdot (\text{it, ot}') \\
& \in \text{Traces}(M) \land \text{it}'(n)[\text{err}] \\
& = \text{false} \land \text{ot}(n)[\text{err}] = \text{true}.
\end{align*}
\]

(15)

We can now state the following theorem, which states that a contract constructed with the above operations holds indeed for any program fulfilling the original contract with an aspect applied to it.

Theorem 1. Let P be a program and let (A, G) be a contract. Then

\[
P \models (A, G)P \triangleleft \text{asp} \\
P \models (\text{Det}_A(\text{OBS}_A(\text{ND}_A(A) \triangleleft \text{asp}))), \text{ Det}_G(\text{OBS}_G(\text{ND}_G(G) \triangleleft \text{asp}))).
\]

(16)

Theorem 1 first transforms the assumption and the guarantee into NDA with the respective operators, then applies the aspect to both and transforms the result back in observers, which are determined. We prove it in Appendix A.

4. EXAMPLE: THE TRAMWAY DOOR CONTROLLER

We implement and verify a larger example, taken from the Lustre tutorial [13], a controller of the door of a tramway. The door controller is responsible for opening the door when the tram stops and a passenger wants to leave the tram, and for closing the door when the tram wants to leave the station. Doors may also include a gangway, which can be extended to allow passengers in wheelchairs enter and leave the tram.

We implement the controller as an Argos program. We first develop a controller for a door without the gangway, and then add the gangway part with aspects. Table 1 gives the in- and outputs of the controller with their specifications, and also the in- and outputs which are added by the gangway.

The controller uses additional inputs, called Helper Signals, which are also shown in Table 1 and are calculated from the original inputs.

It is important for the safety of the passengers that the doors are never open outside a station. We call this property PSafety, and formally express it as an observer that emits err whenever doorClosed-inStation is true. To formally verify this property, we must first develop a model that describes the possible behavior of the physical environment of the controller, which consists of the door and the tramway. These models are also expressed as Argos observers. The models for the tramway (called MTram) and the door (called MDoor) are shown in Figures 9 and 10, respectively. These models require that the environment behaves correctly (e.g., the door only opens if openDoor has been emitted).

Furthermore, we give a contract for the door controller, which focuses on PSafety. The guarantee GContr of the contract is shown in Figure 11, it ensures that the controller emits doorOK only if the doors are closed, and openDoor only if the tram is in a station. The contract has also an assumption AContr, which is the model of the door given in Figure 10, that is, AContr = MDoor. An implementation IContr of the controller, which fulfills the contract, is given in Figure 12.

We can now prove that the controller satisfies the contract (IContr \models (AContr, GContr)), and that the contract in the environment never violates the safety property. Formally, this is expressed as MDoor || MTram || GContr \models PSafety, where the synchronous product of observers means that the properties expressed by all the observers must be fulfilled.

4.1. Adding the gangway

Two aspects are used to add support for the gangway: one aspect Aext that extends the gangway before the door is opened if a passenger has asked for it, and one aspect Aret
Table 1: The interfaces of the controller and the gangway, and the helper signals.

<table>
<thead>
<tr>
<th>Controller Inputs</th>
<th>Controller Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>inStation</td>
<td>doorOK</td>
</tr>
<tr>
<td>leaving</td>
<td>doorClosed</td>
</tr>
<tr>
<td>doorOpen</td>
<td>openDoor</td>
</tr>
<tr>
<td>doorClosed</td>
<td>closeDoor</td>
</tr>
<tr>
<td>askForDoor</td>
<td>beep</td>
</tr>
<tr>
<td>timer</td>
<td>setTimer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gangway Inputs</th>
<th>Gangway outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>gwOut</td>
<td>extendGW</td>
</tr>
<tr>
<td>gwIn</td>
<td>retractGW</td>
</tr>
<tr>
<td>askForGW</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Helper Signals Outputs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptReq</td>
<td>the passenger can ask for the door or the gw</td>
</tr>
<tr>
<td>doorReq</td>
<td>the passenger has asked for the door to open</td>
</tr>
<tr>
<td>gwReq</td>
<td>the passenger has asked for the gangway</td>
</tr>
<tr>
<td>depImm</td>
<td>the tramway wants to leave the station</td>
</tr>
</tbody>
</table>

Figure 10: Model of the door, $M_{Door}$.

Figure 11: The guarantee of the contract of the controller, $G_{Contr}$.

4.2. Modularly verifying the safety properties

We want to check that the new controller $I_{Contr} \triangleleft A_{ext} \triangleleft A_{ret}$ still verifies the safety property from above, and also verifies two new safety properties, which require that the gangway is always fully retracted while the tram is out of station, and that the gangway is never moved when the door is not closed. We express these three properties as an observer and call it $P_{Safeties}$. To verify this, we first weave the aspects into the controller, and thus obtain a new contract that holds for controller with the aspects. Then we check that the environment, to which we added a model of the gangway $M_{GW}$, satisfies the new assumption (i.e., $M_{Door} || M_{Tram} || M_{GW} \models A_{Contr} \triangleleft A_{ext} \triangleleft A_{ret}$), and that the new guarantee satisfies the safety requirements in the environment (i.e., $M_{Door} || M_{Tram} || M_{GW} || G_{Contr} \triangleleft A_{ext} \triangleleft A_{ret} \models P_{Safeties}$).
An alternative to this modular approach is to verify directly that the sample controller with the aspects does not violate the given safety properties (i.e., $M_{\text{Door}} \parallel M_{\text{Tram}} \parallel M_{\text{GW}} \parallel I_{\text{Contr}} \prec A_{\text{ext}} \prec A_{\text{ret}} \models \mathcal{P}_{\text{Safeties}}$). One disadvantage of the alternative approach is that the woven controller may be much bigger than the woven contract. To illustrate this problem, we verified the safety properties using our implementation [14]. The source code of the door controller example is available at [15]. Verifying the woven program takes 11.0 seconds. On the other hand, weaving the aspects into the guarantee of the controller contract and verifying against the environment takes 3.7 seconds, and verifying that the sample controller verifies the contract and verifying that the environment fulfills the assumption with the aspects takes <0.5 second. (Experiments were conducted on an Intel Pentium 4 with 2.4 GHz and 1 Gigabyte RAM.) Thus using this modular approach to verify the safety properties of the controller is significantly faster than verifying the complete program. Although the size of the woven controller is not prohibitive in this example, this indicates that larger programs can be verified using the modular approach.

5. ASPECT INTERFERENCE

A key point when dealing with aspects is the notion of interferences, which is closely related to the way aspects are woven. We illustrate the problem of interfering aspects on an example presented in Section 5.1. Next, we also present a new weaving algorithm in Section 5.2, that weaves aspects jointly, and removes aspect interferences in many cases. Finally, we introduce an algorithm in Section 5.3 that proves noninterference of aspects or identifies remaining interferences in jointly woven programs.

5.1. Example

As an example, we present a simplified view of the interface of a complex wristwatch, implemented with Argos and Larissa. The full case study was presented in [16]. The interface is a modified version of the Altimax model by Suunto. (Suunto and Altimax are trademarks of Suunto Oy.)

5.1.1. The watch

The Altimax wristwatch has an integrated altimeter, a barometer and four buttons, the mode, the select, the plus, and the minus button. Each of the main functionalities (time keeping, altimeter, barometer) has an associated main mode, which displays information, and a number of sub-modes, where the user can access additional functionalities. An Argos program that implements the interface of the watch is shown in Figure 14. For better readability, only those state names, outputs and transitions we will refer to are shown. The buttons of the watch are the inputs of the program. The mode button circles between modes, the select button selects the submodes. There are two more buttons: the plus and the minus button which modify current values in the submodes, but their effect is not shown in the figure. The buttons have different meanings depending on the mode in which the watch is currently.

The interface component we model here interprets the meaning of the buttons the user presses, and then calls a corresponding function in an underlying component. The
outputs are commands to that component. For example, whenever the program enters the time mode, it emits the output Time-Mode, and the underlying component shows the time on the display of the watch.

5.1.2. Two shortcut aspects

The plus and the minus buttons have no function consistent with their intended meaning in the main modes: there are no values to increase or decrease. Therefore, they are given a different function in the main modes: when one presses the plus or the minus button in a main mode, the watch goes to a certain submode. The role of the plus and minus buttons in the main modes are called shortcuts since it allows to quickly activate a functionality, which would have needed, otherwise, a long sequence of buttons.

Pressing the plus button in a main mode activates the logbook function of the altimeter, and pressing the minus button activates the 4-day memory of the barometer. These functions are quite long to reach without the shortcuts since the logbook is the third submode of the altimeter, and the 4-day memory is the second submode of the barometer.

These shortcuts can be implemented easily with Larissa aspects. Figure 15(a) shows the pointcut for the logbook aspect, and Figure 15(b) the pointcut for the memory aspect. In both pointcuts, state main represents the main modes and state sub represents the submodes. When, in a main mode, plus (resp., minus) is pressed, the pointcut emits JP1 (resp., JPm), thus the corresponding advice is executed; when select is pressed, the pointcut goes to the sub state, and JPi or JPm are no longer emitted. Furthermore, we use tolnit advice with traces leading to the functionality we want to reach, that is, σi = mode.select.mode.mode for the logbook aspect and σm = mode.mode.select.mode for the 4-day memory aspect, and the output that tells the underlying component to display the corresponding information.

5.2. Applying several aspects

If we apply first the logbook aspect, and then, sequentially, the memory aspect to the watch program, the aspects do not behave as we would expect. If, in the woven program, we first press the minus button in a main mode, thus activating the logbook aspect, and then the plus button, the memory aspect is activated, although we are in a submode. This behavior was clearly not intended by the programmer of the memory aspect.

The problem is that the memory aspect has been written for the program without the logbook aspect: the pointcut assumes that the only way to leave a main mode is to press the select button. However, the logbook aspect invalidates that assumption by adding transitions from the main modes to a submode. When these transitions are taken, the pointcut of the memory aspect incorrectly assumes that the program is still in a main mode.

Furthermore, for the same reason, applying first the memory aspect and then the logbook aspect produces (in terms of trace-equivalence) a different program from applying first the logbook aspect and then the memory aspect: watch ∼ logbook ∼ memory ∼ watch ∼ memory ∼ logbook.

As a first attempt to define aspect interference, we say that two aspects A1 and A2 interfere when their application on a program P in different orders does not yield two trace-equivalent programs: P ∼ A1 ∼ A2 ∼ P ∼ A2 ∼ A1. We say that two aspects that do not interfere are independent.

With interfering aspects, the aspect that is woven second must know about the aspect that was applied first. To be able to write aspects as the ones above independently from each other, we propose a mechanism to weave several aspects at the same time. The idea is to first determine the join point transitions for all the aspects, and then apply the advice.

Definition 16 (Joint weaving of several aspects). Let A1 · · · An be some aspects, with Ai = (PJi, advi), and P a program. We define the application of A1 · · · An on P as follows:

\[ P ∼ (A1, ..., An) = P \cdot PJ1 \cdot ... \cdot PJn \cdot \text{adv1} \cdot ... \cdot \text{advn} \]  

(17)

Note that Definition 16 reuses the advice weaving operator defined in Definition 5, and indexes the join point signal used by each advice. Furthermore, the advice is woven in the reverse order, that is, we first the advice from the last aspect in the aspect list, and the advice from the first aspect last. This way, aspects that are later in the list have higher priority: if a join point transition is claimed by several aspects, the one that is woven first replaces the join point transition with its advice transition, and removes the join point signals of the other aspects. To give priority to the aspects that are applied later is consistent with sequential weaving, where aspects that are applied later modify the aspects that have already been applied, but not the other way round.

Jointly weaving the logbook and the memory aspect leads to the intended behavior, that is, both aspects can be activated only when the program is in a main mode. Furthermore, the weaving order does not influence the result because both aspects first select their join point transitions.
in the main modes, and change the target states of the join point transitions only afterwards.

Note that Definition 16 does not make sequential weaving redundant. We still need to weave aspects sequentially in some cases, when the second aspects must be applied to the result of the first. For instance, imagine an aspect that adds redundant. We still need to weave aspects sequentially in the main modes, and change the target states of the join point transitions only afterwards.

Definition 16 does not solve all conflicts. Indeed, the $A_i$ in $P \prec (A_1, \ldots, A_n)$ do not commute, in general, since the advice weaving is applied sequentially. We define aspect interference for the application of several aspects.

Definition 17 (Aspect interference). Let $A_1 \cdot \cdot \cdot A_n$ be some aspects, and $P$ a program. We say that $A_i$ and $A_{i+1}$ interfere for $P$ if and only if

$$P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n) \not\sim P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n).$$

(18)

As an example for interfering aspects, assume that the condition of the join point transition of the pointcut of the logbook aspect (Figure 15(a)) is only minus and the condition of the join point transition of the pointcut of the logbook aspect (Figure 15(b)) is only plus. In this case, the two aspects share some join point transitions, namely, when both buttons are pressed at the same time in a main mode. Both aspects then want to execute their advice, but only one can, thus they interfere. Only the aspect that was applied last is executed.

In such a case, the conflict should be made explicit to the programmer, so that it can be solved by hand. Here, it was resolved by changing the pointcuts to the form they have in Figure 15, so that neither aspect executes when both buttons are pressed.

5.3. Proving noninterference

In this section, we show that in some cases, noninterference of aspects can be proven, if the aspects are woven jointly, as defined in Definition 16. We can prove noninterference of two given aspects either for any program, or for a given program. Following [17], we speak of strong independence in the first case, and of weak independence in the second.

We use the operator $\text{jpTrans}$ to determine interference between aspects. It computes all the join point transitions of an automaton, that is, all transitions with a given output $JP$.

Definition 18. Let $A = (Q, s_{init}, I, \Theta, T)$ be an automaton and $JP \in \Theta$. Then

$$\text{jpTrans}(A, JP) = \{ t \mid t = (s, \ell, O, s') \in T \land JP \in O \}. \quad (19)$$

The following theorem proves strong independence between two aspects.

Theorem 2 (strong independence). Let $A_1 \cdot \cdot \cdot A_n$ be some aspects, with $A_i = (P_{\ell P_i, \text{adv}})$. Then the following equation holds:

$$\text{jpTrans}(P_{\ell P_i, JP_i}) \cap \text{jpTrans}(P_{\ell P_{i+1}, JP_{i+1}}) = \emptyset$$

$$P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n)$$

$$\sim P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n).$$

(20)

See Appendix B for a proof. Theorem 2 states that if there is no transition with both $JP_i$ and $JP_{i+1}$ as outputs in the product of $P_{\ell P_i}$ and $P_{\ell P_{i+1}}$, $A_i$ and $A_{i+1}$ are independent and thus can commute while weaving their advice. Theorem 2 defines a sufficient condition for noninterference, by looking only at the pointcuts. When the condition holds, the aspects are said to be strongly independent.

Theorem 3 (Weak independence). Let $A_1 \cdot \cdot \cdot A_n$ be some aspects, with $A_i = (P_{\ell P_i, \text{adv}})$, and $P_{\ell c} = \mathcal{P}(P, P_{\ell P_1}, \ldots, P_{\ell P_n})$. Then the following equation holds:

$$\text{jpTrans}(P_{\ell c}, JP) \cap \text{jpTrans}(P_{\ell c}, JP_{i+1}) = \emptyset$$

$$P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n)$$

$$\sim P \prec (A_1 \cdot \cdot \cdot A_i, A_{i+1} \cdot \cdot \cdot A_n).$$

(21)

See Appendix C for a proof. Theorem 3 states that if there is no transition with both $JP_i$ and $JP_{i+1}$ as outputs in $P_{\ell c}$. $A_i$ and $A_{i+1}$ do not interfere. This is weaker than Theorem 2 since it also takes the program $P$ into account. However, there are cases in which the condition of Theorem 2 is false (thus it yields no results), but Theorem 3 allows to prove noninterference, for example, in the case of the gangway aspects from Section 4, which is discussed in Section 5.3.2.

Theorem 3 is a sufficient condition, but, as Theorem 2, it is not necessary: it may not be able to prove independence for two independent aspects. The reason is that it does not take into account the effect of the advice weaving: consider two aspects such that the only reason why the condition for Theorem 3 is false is a transition sourced in some state $s$, and such that $s$ is only reachable through another join point transition; if the advice weaving makes this state unreachable, then the aspects do not interfere.

The results obtained by both Theorems are quite intuitive. They mean that if the pointcuts do not select any join points common to two aspects, then these aspects do not interfere. This condition can be calculated on the pointcuts alone, or can also take the program into account.

Note that the detection of noninterference is a static condition that does not add any complexity overhead. Indeed, to weave the aspects, the compiler needs to build first $P_{\ell P_1, \ldots, \ell P_n} = P_{\ell c}$: the condition of Theorem 2
can be checked during the construction of $P_{allJP}$. Second, the weaver builds $P_{pc} = P(P_{allJP})$. Afterwards, it can check the condition of Theorem 3. Thus to calculate the conditions of both Theorems, it is sufficient to check the outputs of the transitions of intermediate products during the weaving. The weaver can easily emit a warning when a potential conflict is detected.

To have an exact characterization of noninterference, it is still possible to compute the predicate $P \triangleleft (A_1 \cdots A_{i+1} \cdots A_n) ~ \sim P \triangleleft (A_1 \cdots A_{i+1}, A_i \cdots A_n)$, but calculating semantic equality is very expensive for large programs.

Note that the interference presented here only applies to the joint weaving of several aspects, as defined in Definition 16. Sequentially, woven aspects may interfere even if their join points are disjoint, because the pointcut of the second aspects applies to the woven program. A similar analysis to prove noninterference of sequential weaving would be more difficult because the effect of the advice must be taken into account. Indeed, the advice of an aspect influences which transitions are selected by the pointcut of an aspect that is sequentially woven next.

5.3.1. Interference between the shortcut aspects

Figure 16(a) shows the product of the pointcuts of the logbook (Figure 15(a)) and the memory aspect (Figure 15(b)). There are no transitions that emit both $JP_l$ and $JP_m$, thus, by applying Theorem 2, we know that the aspects do not interfere, independently of the program they are applied to.

Let us assume again that the condition of the join point transition of the pointcut of the logbook aspect (Figure 15(a)) is only $\sim$ and the condition of the join point transition of the pointcut of the logbook aspect (Figure 15(b)) is only $\triangleright$. In this case, the state main in Figure 16(a) would have another loop transition, with label $\sim$ plus $\triangleright$. Thus, Theorem 2 not only states that the aspects potentially interfere, but it also states precisely where: here, the problem is that when both $\sim$ and $\triangleright$ are pressed in a main mode, at the same time, both aspects are activated. Larissa thus emits a warning and the user can solve the conflict.

5.3.2. Interference between the gangway aspects

As an example for weak interference, let us examine the gangway aspects from the Tram example in Section 4. Figure 16(b) shows the product of their pointcuts. There is a transition that has both $JP_{ext}$ and $JP_{ret}$ as outputs. Theorem 2 states that the aspects may interfere, but when applied to the tram controller from Figure 12, they do not. This is because $door\overline{OK}$ and $openDoor\overline{a}$ are outputs of the controller, and are never emitted at the same time.

In this example, the use of Theorem 3 is thus needed to show that the aspects do not interfere when applied to the wristwatch controller. As expected, $JP_{ext}$ and $JP_{ret}$ are never emitted at the same time in $P_{pcv}$, and Theorem 3 thus shows that the aspects do not interfere for this base program.

6. RELATED WORK

6.1. Contracts and aspects

Goldman and Katz [18] modularly verify aspect-oriented programs using an LTL tableau representation of programs and aspects. As opposed to ours, their system can verify AspectJ aspects, as tools like Bandera [19] can extract suitable input models from Java programs. It is, however, limited to the so-called weakly invasive aspects, which only return to states already reachable in the base program.

Clifton and Leavens [20] noted before us that aspects invalidate the specification of modules, and propose that either an aspect should not modify a program’s contract, or that modules should explicitly state which aspects may be applied to them.

6.2. Aspect interference

Some authors discuss the advantages of sequential versus joint weaving. Lopez-Herrejon and Batory [21] propose to use sequential weaving for incremental software development. Colyer and Clement [22, Section 5.1] want to apply aspects to bytecode which already contains woven aspects. In AspectJ, this is impossible because the semantics would not be the same as weaving all aspects at the same time.

Sihman and Katz [23] propose SuperJ, a superimposition language which is implemented through a preprocessor for AspectJ. They propose to combine superimpositions into a new superimposition, either by sequentially applying one to the other, or by combining them without mutual influence. Superimpositions contain assume/guarantee contracts, which can be used to check if a combination is valid.

A number of authors investigate aspect interference in different formal frameworks. Much of the work is devoted to determining the correct application order for interfering aspects, whereas we focus on proving noninterference.

Douence et al. [17] present a mechanism to statically detect conflicts between aspects that are applied in parallel. Their analysis detects all join points where two aspects want to insert advice. To reduce the detection of spurious conflicts, they extend their pointcuts with shared variables, and add constraints that an aspect can impose on a program. To resolve remaining conflicts, the programmer can then write powerful composition adaptors to define how the aspects react in presence of each other.

Pawlak et al. [24] present a way to formally validate precedence orderings between aspects that share join points. They introduce a small language, CompAr, in which the user expresses the effect of the advice that is important for aspect interaction, and properties that should be true after the execution of the advice. The CompAr compiler can then check that a given advice ordering does not invalidate a property of an advice.

Dur et al. [25] propose an interaction analysis for composition filters. They detect when one aspect prevents the execution of another, and can check that a specified trace property is ensured by an aspect.
Balzarotti et al. [26] use program slicing to check if different aspects modify the same code, which might indicate interference.

7. CONCLUSION

We presented two formal analysis tools for Larissa, which both exploit its semantic definition. The first combines Larissa with design-by-contract, and shows exactly how a Larissa aspect modifies the contract of a component to which it is applied. This allows us to calculate the effect of an aspect on a specification instead of only on a concrete program. This approach has several advantages. First, aspects can be checked against contracts even if the final implementation is not yet available during development. Furthermore, if the base program is changed, the woven program must not be reverified, as long as the new base program still fulfills the contract. Finally, woven programs can be verified modularly, which may allow to verify larger programs.

The second approach is an analysis for aspect interference in Larissa. We introduced an additional operator which jointly weaves several aspects together into a program, closer to the way AspectJ weaves aspects. Because Larissa is defined modularly, we only had to rearrange the building steps of the weaving process. Then we could analyze interference with a simple parallel product of the pointcuts. When a potential conflict is detected, the user has to solve it by hand, if needed. In the examples we studied, the conflicts were solved by simple modifications of the pointcuts.

These analyses are only possible because Argos and Larissa are very simple languages with clean and simple semantics. They thus illustrate the advantage of using a programming language with simple semantics. Because of this simplicity, both approaches seem quite precise. Indeed, we believe that the contract weaving is exact in that it gives no more possible behaviors for the woven program than necessary. That is, for a contract C and a trace \( t \in \text{Traces}(C \triangleleft \text{ASP}) \), there exists a program \( P \) s.t. \( P \models C \) and \( t \in \text{Traces}(P \triangleleft \text{ASP}) \). This remains however to be proven.

The interference analysis for Larissa seems quite precise, that is, we can prove independence for most independent aspects. One reason for that are Larissa’s powerful pointcuts, which describe join points statically, yet very precisely, on the level of transitions. Another reason is the exclusive nature of the advice. Two pieces of advice that share a join point transition never execute sequentially, but there is always one that is executed while the other is not. If the two pieces of advice are not equivalent, this leads to a conflict. Thus as opposed to [17], assuming that a shared join point leads to a conflict does not introduce spurious conflicts.

There are some interesting points for further work. In the context of contract weaving, an interesting question is if we can derive contracts the other way round. Given a contract \( C \) and an aspect \( \text{ASP} \), can we automatically derive a contract \( C' \) such that \( C' \triangleleft \text{ASP} \models C \)? Finally, both approaches work only because we have restricted Argos and Larissa to Boolean signals. It would be interesting to see if they can be extended to programs with variables.

APPENDICES

A. PROOF FOR THEOREM 1

Definitions

We first introduce a number of definitions.

\( P(p) \models (A(a), G(g)) \) means that program \( P \) fulfills contract \( (A, G) \), where the initial states of \( P, A, \) and \( G \) have been set to \( p, a, \) and \( g, \) respectively.

Furthermore, we introduce the following notations for terms from the theorem. Let

\[
\begin{align*}
A' \triangleleft \text{ASP} & = \text{OBS}_{A}(\text{ND}_{A}(A) \triangleleft \text{ASP}), \\
A \triangleleft \text{ASP} & = \text{Det}_{A}(A' \triangleleft \text{ASP}), \\
G' \triangleleft \text{ASP} & = \text{OBS}_{G}(\text{ND}_{G}(G) \triangleleft \text{ASP}), \\
G \triangleleft \text{ASP} & = \text{Det}_{G}(G' \triangleleft \text{ASP}).
\end{align*}
\]

We now define the structure of some of these terms. Let

\[
\begin{align*}
P & = (Q_P, q_{P0}, I, \Theta, \mathcal{T}_P), \\
\text{ASP} & = (PC, \langle O_{adv}, \text{tolInit}, \sigma \rangle), \\
PC & = (Q_{PC}, q_{PC0}, I \cup \Theta, \{JF\}, \mathcal{T}_{PC}), \\
A & = (Q_A \cup \{\text{Error}\}, q_{A0}, I \cup \Theta, \{\text{err}\}, \mathcal{T}_A), \\
G & = (Q_G \cup \{\text{Error}\}, q_{G0}, I \cup \Theta, \{\text{err}\}, \mathcal{T}_G), \\
P & \triangleleft \text{ASP} = ((Q_P \times Q_{PC}), (q_{P0}, q_{PC0}), I, \Theta, \mathcal{T}_{P\triangleleft}), \\
A' & \triangleleft \text{ASP} = ((Q_A \times Q_{PC}), \{\text{Error}\}, q_{AB}, q_{PC0}), \\
\text{ASP} & = (I \cup \Theta, \{\text{err}\}, \mathcal{T}_{A\triangleleft}), \\
G' & \triangleleft \text{ASP} = ((Q_G \times Q_{PC}), \{\text{Error}\}, q_{GB}, q_{PC0}), \\
\text{ASP} & = (I \cup \Theta, \{\text{err}\}, \mathcal{T}_{G\triangleleft}).
\end{align*}
\]

We prove the theorem by induction over a trace of \( P \triangleleft \text{ASP} \). Let \((it, ot) \in \text{Traces}(P \triangleleft \text{ASP})\). We show that the following induction hypothesis holds for any \( n \).
**Induction hypothesis**

The induction hypothesis states that the states reached by executing \((it, ot)\) on \(P < asp\), \(A' < asp\), and \(G' < asp\) formed a valid contract in \(P, A, G\), that is, before the aspect was applied, provided \((it, ot)\) is accepted by \(A < asp\). Formally, we write it as follows:

\[
\text{O}_{step}^{A < asp}(s_{A0}, s_{PC0}), it, ot, n) = \emptyset \land (p_n, p_{cn})
\]

\[
\exists (a_n, p_{cn}) \in S_{step}^{A' < asp}(s_{A0}, s_{PC0}), it, ot, n)
\]

\[
\exists (g_n, p_{cn}) \in S_{step}^{G' < asp}(s_{G0}, s_{PC0}), it, ot, n)
\]

\[
P_n = (A(a_n), G(g_n)) \land g_n \neq Error
\]

(A.3)

\((p_n, p_{cn}), (a_n, p_{cn})\), and \((g_n, p_{cn})\) are the states reached when executing \((it, ot)\) for \(n + 1\) steps on \(P < asp\), \(A' < asp\) and \(G' < asp\), respectively.

**Base case**

\(n = 0\). \(P \models (A, G)\) holds as it is the assumption of the implication in the theorem. If the initial state of \(G\) is the error state, either \(A\) and \(A < asp\) do not accept any trace, or no \(P\) exists, and in both cases we are done.

**Induction step**

From \(n \to n + 1\).

If \(O_{step}^{A < asp}(it, ot, n) = \{\{\text{Error}\}\}\), we are done. Otherwise, \(O_{step}^{A < asp}(it, ot, n) = \{\emptyset\}\) follows \(O_{step}^{A' < asp}(it, ot, n) = \{\emptyset\}\) because Definition 14, which defines the determination of \(A\), gives precedence to error transitions. We distinguish two cases as follows:

(i) First case: \([J]\) does not contain a join point.

Because of \(P(p_{n-1}) = (A(a_{n-1}), G(g_{n-1}))\), there is a transition \(t_{P} = (p_{n-1}, it, n), (o_{n}, p_{a})\) in \(T_{P}\), a transition \(t_{A} = (a_{n-1}, it, n), (o_{n}, a)\) in \(T_{A}\), and a transition \(t_{G} = (g_{n-1}, it, n), (o_{n}, g)\) in \(T_{G}\), such that \(P(p_{n}) = (A(a_{n}), G(g_{n})), t_{P}, a\) and \(t_{G}\) are not modified by the weaving, thus there is a transition \((\langle p_{n-1}, p_{cn-1}, it, n), (o_{n}, p_{a})\) in \(T_{PC}\), a transition \((\langle a_{n-1}, p_{cn-1}, it, n), (o_{n}, a)\) in \(T_{A}\), \(T_{A}\) and \(T_{G}\) in \(T_{G}\), with \((g_{n}, p_{cn})\) in with \((g_{n}, p_{cn})\) in \(T_{G}\).

(ii) Second case: \([J]\) does not contain a join point.

Let \(p_{a} = S_{step}^{P}(s_{P0}, \sigma_{i}, l_{a})\) be the state in \(P\) reached after executing \(a\), and \(\sigma_{i}\), then a trace of length \(l_{a}\) such that for all \(i \leq l_{a}\), \(S_{step}^{P}(s_{P0}, \sigma_{i}, l_{a})\) is defined.

All join point transitions in \(G' < asp\) (resp., \(A' < asp\)) are replaced by transitions to all possible target states, such as there is a transition \(t_{G} \in T_{G}\) (resp., \(t_{A} \in T_{A}\)) to a target state \((g_{n}, p_{cn})\) (resp., \(a_{n}, p_{cn}\)) such that \(S_{step}^{G}(s_{G0}, \sigma_{i}, l_{a}) = g_{n}\) (resp., \(S_{step}^{A}(s_{A0}, \sigma_{i}, l_{a}) = a_{n}\)).

Because \(p_{a}, a_{n}, g_{n}\) can be reached with the same trace \((\sigma_{i}, \sigma_{o}, \sigma_{a})\) for \(P\) and \(G\) from the initial state, \(P(p_{0}) = (A(a_{0}), G(g_{0}))\) follows from \(P \models (A, G)\).

Furthermore, \((o_{n}) = \epsilon_{G_{0}} \land \epsilon_{P_{0}}\), and we have \(t_{G} \in (\langle a_{n-1}, p_{cn-1}, it, n), (o_{n}), (a_{n}, p_{cn}), (g_{n-1}, p_{cn-1}, it, n), (o_{n}), (g_{n}, p_{cn})\), and thus \((a_{n}, p_{cn}) = S_{step}^{A' < asp}(s_{A0}, s_{PC0}), it, ot, n)\) and \((g_{n}, p_{cn}) = S_{step}^{G' < asp}(s_{G0}, s_{PC0}), it, ot, n)\). Furthermore, we have \((g_{n}, p_{cn}) \neq Error\), as otherwise \(a_{n} = Error\) (impossible because of \(O_{step}^{A' < asp}(s_{A0}, s_{PC0}), it, ot, n) = \emptyset\), or \((it, ot) \in Traces(P)\), by the definition of \(P \models (A, G)\).

It follows from the induction hypothesis that

\[
(it, ot) \in Traces(A < asp) \land (it, ot) \in Traces(P < asp) \land (it, ot) \in Traces(G' < asp),
\]

and we have \((it, ot) \in Traces(G' < asp) \Rightarrow (it, ot) \in Traces(G < asp)\) by Definition 15. Thus the theorem follows from the induction hypothesis.

**B. PROOF FOR THEOREM 2**

Theorems 2 and 3 are both implications with the same consequent.

We show that the antecedent of the implication in Theorem 3,

\[
(jp Trans(P(P_{P}, P_{P_{+}}), \ldots, P_{P_{N}}), J_{P})
\]

\[
\land j p Trans(P(P_{P}, P_{P_{+}}), \ldots, P_{P_{N}}), J_{P_{+1}}) = \emptyset,
\]

follows from the antecedent in Theorem 2,

\[
(jp Trans(P_{P}, P_{P_{+}}), J_{P})
\]

\[
\land j p Trans(P_{P}, P_{P_{+}}), J_{P_{+1}} = \emptyset,
\]

\(J_{P}, J_{P_{+1}}\) can only occur in \(P_{P}\) and \(P_{P_{+1}}\). Thus if a transition that has both of them as outputs in \(P(P, P_{P}, \ldots, P_{P_{+}})\), there must already exist a transition with both of them as outputs in \(P_{P}, P_{P_{+}}\).

Thus because of the transitivity of the implication, Theorem 2 is a consequence of Theorem 3.

**C. PROOF FOR THEOREM 3**

Because the synchronous product is commutative \(P(P, P_{P}, \ldots, P_{P_{+}}) = \ldots, P_{P_{+}})\) and \(P(P, P, \ldots, P_{P_{+}})\) are the same.

Let \(P(P, P_{P}, \ldots, P_{P_{+}}) < j p_{P}, adv_{1}, \ldots, j p_{P}, adv_{2}\) yield an automaton \(P_{P_{+1}} = (Q, s_{init}, l_{P}, \emptyset, O_{adv} \cup O_{adv_{+}}, T')\), where \(T'\) is defined as follows:

\[
((s, l, O, s') \in T' \land J_{P_{+1}} \notin O) \quad ((s, l, O, s') \in T')
\]

\[
((s, l, O, s') \in T' \land J_{P_{+1}} \in O)
\]

\((s, l, O_{adv_{+}}, S_{step}(s_{init}, s_{+1}, l_{O_{adv_{+}}})) \in T'\).
and $P_{i+1} \prec\prec_{JP, \text{adv}_i} \prec\prec_{JP, \text{adv}_i}$ yields an automaton $P_i = (Q, s_{\text{init}}, I, O \cup O_{\text{adv}}, \cup O_{\text{adv}}, T')$, where $T'$ is defined as follows:

\[
(s, \ell, O, s') \in T \wedge JP_{i+1} \in O \wedge JP_i \notin O)
\]

\[
(s, \ell, O, s') \in T',
\]

\[
((s, \ell, O, s') \in T \wedge JP_{i+1} \in O \wedge JP_i \notin O)
\]

\[
(s, \ell, O_{\text{adv}}, s_{\text{step}}(s_{\text{init}}, s_{\ell}, l_{\ell})) \in T',
\]

\[
((s, \ell, O_{\text{adv}}, s_{\text{step}}(s_{\text{init}}, s_{\ell}, l_{\ell})) \in T',
\]

\[
((s, \ell, O_{\text{adv}}, s_{\text{step}}(s_{\text{init}}, s_{\ell}, l_{\ell})) \in T',
\]

\[
(s, \ell, O_{\text{adv}}, s_{\text{step}}(s_{\text{init}}, s_{\ell}, l_{\ell})) \in T'.
\]

If we calculate $P_{i+2} \prec\prec_{JP, \text{adv}_i} \prec\prec_{JP, \text{adv}_i+1}$, we obtain the same automaton, except for transitions (C.9), which are defined by

\[
((s, \ell, O, s') \in T \wedge JP_{i+1} \in O \wedge JP_i \in O)
\]

\[
(s, \ell, O_{\text{adv}}, s_{\text{step}}(s_{\text{init}}, s_{\ell}, l_{\ell})) \in T'.
\]

Transitions (C.9) are exactly the join point transitions that are in $JP \setminus \text{Trans}(P, P_{JP}, \cdots \| P_{JP}, JP_i \cap JP \setminus \text{Trans}(P, P_{JP}, \cdots \| P_{JP}, JP_{i+1})$. By preconditions, there were no such transitions in $P, P_{JP}, \cdots \| P_{JP}$. Because we require that all the $JP_i$ operations occur nowhere else, $J P_i$ and $J P_{i+1}$ cannot be contained in an $O_{\text{adv}}$, thus no transition of type (C.9) has been added by the weaving of $\prec\prec_{JP, \text{adv}_i} \cdots \prec\prec_{JP, \text{adv}_i+2}$.

Thus we have $P, P_{JP}, \cdots \| P_{JP} \prec\prec_{JP, \text{adv}_n} \prec\prec_{JP, \text{adv}_{n+1}} \prec\prec_{JP, \text{adv}_{n+1}} \prec\prec_{JP, \text{adv}_{n+1}} = P, P_{JP}, \cdots \| P_{JP} \prec\prec_{JP, \text{adv}_n} \prec\prec_{JP, \text{adv}_{n+1}} \prec\prec_{JP, \text{adv}_{n+1}} \prec\prec_{JP, \text{adv}_{n+1}} \prec\prec_{JP, \text{adv}_1}$. Weaving $\prec\prec_{JP, \text{adv}_i} \cdots \prec\prec_{JP, \text{adv}_1}$ trivially yields the same result.

REFERENCES


Research Article

Embedded Systems Programming: Accessing Databases from Esterel

David White and Gerald Lütgen

Department of Computer Science, University of York, Heslington, YorkYO10 5DD, UK

Correspondence should be addressed to Gerald Lütgen, gerald.luetgen@cs.york.ac.uk

Received 18 March 2008; Revised 23 July 2008; Accepted 11 November 2008

Recommended by Alain Girault

A current limitation in embedded controller design and programming is the lack of database support in development tools such as Esterel Studio. This article proposes a way of integrating databases and Esterel by providing two application programming interfaces (APIs) which enable the use of relational databases inside Esterel programs. As databases and Esterel programs are often executed on different machines, result sets returned as responses to database queries may be processed either locally and according to Esterel’s synchrony hypothesis, or remotely along several of Esterel’s execution cycles. These different scenarios are reflected in the design and usage rules of the two APIs presented in this article, which rely on Esterel’s facilities for extending the language by external data types, external functions, and procedures, as well as tasks. The APIs’ utility is demonstrated by means of a case study modelling an automated warehouse storage system, which is constructed using Lego Mindstorms robotics kits. The robot’s controller is programmed in Esterel in a way that takes dynamic ordering information and the warehouse’s floor layout into account, both of which are stored in a MySQL database.

Copyright © 2008 D. White and G. Lütgen. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

One of the current limitations in the programming of embedded controllers is the lack of database support available within languages such as Esterel [1, 2] and Lustre [3], and their development environments, Esterel Studio and SCADE [4], respectively. These environments are used by large avionics manufacturers and vendors of digital signal processing solutions for developing the software of complex, and often safety-critical, embedded systems. Both Esterel and Lustre are synchronous languages which aim at describing reactions in cycle-based reactive systems, including embedded controllers. Such systems continuously interact with their physical environment by (i) reading in signals representing sensor values, such as an aircraft’s speed, altitude, and attitude; (ii) computing a reaction based on these values, such as a rudder angle; (iii) emitting signals carrying the computed values to the environment, that is, to the hydraulic system moving the rudders. While Esterel is a textual, imperative language that aims at modelling control flow and has semantical similarities to Statecharts [5], Lustre is best suited for modelling data flow and is a graphical language centred around block diagrams, very much like Simulink [6].

The problem

What all development environments that are available for these languages have in common is that they support the automatic generation of code, such as C, Ada, or VHDL code, from abstract program descriptions. In this way, they aim to make embedded software design and programming more cost-effective when compared to traditional software development processes. However, Esterel Studio and SCADE do not provide an easy way of integrating databases within an application. Other reactive systems design tools are very limited in this respect as well, including Simulink/Stateflow [7] and Statemate [8, 9]. As is, a system designer needs to modify autogenerated code by hand in order to interface to databases, which is both difficult and error-prone. This is a problem very much relevant in industry since some reactive systems programmed in synchronous languages would benefit from an easy model of database interaction. For
example, synchronous languages are often used to build the flight software for aeroplanes. Adding database interaction would enable spatial and mapping data to be retrieved and processed directly by the reactive kernel implementing an autopilot. Further examples are infotainment systems in the automotive sector, particularly navigation systems, or process control systems in nuclear reactors where regulators require that logs of data are recorded and kept.

Our contribution

This article addresses the aforementioned limitation by providing Application Programming Interfaces (APIs) for using relational databases within the Esterel programming language. We choose MySQL [10] as the database and, since reactive kernels are produced as C programs by the Esterel compiler [11–13], the APIs are implemented using the MySQL C interface [14] whose functionality we aim to mirror in our APIs for Esterel. MySQL is selected here simply for its convenience and since it is widely used. However, our approach can as easily be applied to other relational databases. To the best of our knowledge, no work on database integration within Esterel, or similar languages, has been published in the literature before. This does not mean, however, that we are the first to integrate an existing synchronous language with a database system. The problem is that other works are commercial and not in the open domain. This includes National Instruments’ LabVIEW Database Connectivity Toolkit [15] which is a set of tools for connecting programs designed in LabVIEW to popular databases, such as Microsoft Access, SQL Server, and Oracle, and for implementing many common database operations without having to perform SQL programming.

Because database transactions are relatively complex when compared to responses of Esterel reactive kernels, databases and reactive programs must be considered as running asynchronously to each other. This is true regardless of whether they reside on the same machine or on different machines. In the former case, however, result sets to database queries may reasonably be assumed to be processed within a single synchronous step of the reactive kernel. In the latter case, result sets are necessarily read asynchronously to the reactive kernel. For these reasons, one API for each situation is provided; a Local Result Set API and a Remote Result Set API. The realisation of both APIs relies on Esterel’s support for extending the language via external data types, external functions and procedures, and tasks. For example, the Local Result Set API makes heavy use of external functions and procedures. This is because these are considered to execute instantaneously, that is, within a single reaction cycle, and therefore do not interfere with the synchronous nature of Esterel programs. On the other hand, the Remote Result Set API, that houses a microcomputer capable of running an Esterel reactive kernel. Sensors and actuators connected to the RCX, such as touch, light and rotation sensors, and motors, respectively, permit interaction with the RCX’s environment. The RCX also has a built-in infrared port, which we use to communicate with the warehouse database on the server.

Organisation

The next section gives a brief introduction to the Esterel language. Section 3 describes both our APIs, emphasising the general model of interaction between Esterel reactive kernels and databases. Some details regarding the APIs’ implementations can be found in Section 4. Our case study involving the warehouse storage system is presented in Section 5, while Section 6 contains our conclusions.

2. A BRIEF OVERVIEW OF ESTEREL

The Esterel language for programming embedded controllers has been developed by Gérard Berry in France since the early eighties [18] and has been commercialised by Esterel Technologies in their Esterel Studio design suite [4].

Esterel is part of a family of languages—the so called synchronous languages—that are reactive and synchronous. Reactive means that an Esterel program is constantly interacting with its environment, while synchronous means that these interactions are periodic, with the computation of each reaction being “instantaneous”, that is, reactions are computed faster than the environment is sampled for inputs. Esterel affects and samples its environment through signals, which are its primary communication device. For example, a signal representing a button press will either be present if the button is pressed or absent if not. Signals can also carry data values, which is useful when working with sensor readings and actuator parameters.

Moreover, Esterel is an imperative programming language. It is particularly suited for control-dominated applications as it allows for the expression of parallelism and preemption. Its core language elements include the following statements [1]:

(i) emit S emits signal S in the current instant, that is, the current reaction cycle;
(ii) present S then stmt1 else stmt2 end checks whether signal S can be determined as either present or absent in the current instant; if it is present, then stmt1 is executed, and if it is absent, then control passes to stmt2;
We illustrate Esterel's syntax and semantics by means of the small example; note that this example is not intended to have any particular meaning:

```plaintext
output X : integer;
input A1, A2;
input B : boolean;

module test_program:
  emit X(10);
  loop
    await A1 || await A2;
    await immediate B;
    pause;
    if ?B then
      present A1 then
        emit X(100);
    end
  end
end module
```

Our example program has one output signal X carrying an integer value, two input signals A1 and A2 carrying no values, and one input signal B carrying a Boolean value. As can be seen, Esterel programs are split into modules, so as to support the concepts of program decomposition and software reuse.

In the first instant, that is, reaction cycle, of our example program, value 10 is emitted on output signal X, and the loop body is entered. This is followed by the await A1 || await A2 statement which tells the program to concurrently wait for input signals A1 and A2, and ends the first instant. The program now continues waiting for A1 and A2 forever, so let us assume that these signals are received from the environment sometime before the second instant. Therefore, in the second instant, statement await A1 || await A2 completes and await immediate B is executed. The use of the immediate keyword means that the input signal can arrive in the instant that executed the await immediate statement, that is, it does not force the next instant to take place like statement await. Instead, a pause statement can be used in order to force the next instant to occur. The present statement shows how to access values carried via signals, namely through the use of the ? operator. Hence, if the value of signal B read in the previous instant is true and if input signal A1 is present, then the integer 100 is output via signal X, before the loop executes again.

The semantics of Esterel is well defined and has been extensively investigated in the literature [2]. In addition to the synchrony hypothesis which underlies the concept of cycle-based reaction, the semantics is based on the principles of consistency, causality, reactivity, and determinism. Consistency means that a signal cannot be both present and absent within the same instant. Every presence and absence of a signal must further be causally justified, by ultimately referring to the presence and absence of the input signals. Moreover, a program must permit a reaction, no matter what the statuses and values of the input signals are, that is, it must be reactive. It must also compute a unique reaction in each instant, for each possible input, whence it is deterministic.

It has been mathematically verified that Esterel’s semantics possesses many desirable properties. In particular, the set of all valid Esterel programs corresponds one-to-one to the set of those asynchronous digital circuits with feedback that stabilise independently of any gate or wire delays [19]. This close relationship between programs and circuits is utilised by the code generators available in Esterel Studio. While creating VHDL or Verilog from a valid Esterel program involves synthesising the circuit corresponding to the program, Esterel Studio’s C code generator essentially simulates this synthesised circuit.

Last, but not least, it must be pointed out that Esterel is an extensible language, which allows users to define external data types, external functions, and procedures, as well as tasks. Given that the processing of reactions must be quicker than the system’s environment, it seems reasonable to disallow any operations to take place during the kernel processing that might cause delay to a subsequent instant. It is for this reason that external functions and procedures in Esterel must be instantaneous [1]. If asynchronous execution of external code is desired, then tasks ought to be used. We have made extensive use of Esterel’s extensibility features during the development of our database APIs.

### 3. DATABASE APIs FOR ESTEREL

In this section, we devise two different APIs for enabling relational database access within Esterel, depending on whether result sets to queries are stored locally or remotely to the Esterel reactive kernel. Both allow multiple, simultaneous connections to databases and are intended for use in different application scenarios.

We start off by providing the rationale for developing two APIs, for which we review the possible computing architectures running an Esterel reactive kernel and a database. Both our APIs consider databases as part of the system environment and as running asynchronously to the reactive kernel. This is because database transactions are typically more complex to process than ordinary reactions.

A typical database interaction consists of several stages. First, the database is queried and a result set is generated according to that query. Such a result set is simply a set of rows, possibly ordered, that contains the data specified in the query. From this point on, most database management systems allow for two different routes to access the result set. One method is to transfer the whole result set to the client that issued the
query, and let the client use the information as necessary. The alternative method is for use in situations where it is infeasible to transfer the whole result set, due to memory or bandwidth constraints. Instead, the client machine may access the result set stored on the database server in a row-by-row manner, minimising the speed at which data needs to be transferred to the client machine. This dual-retrieval method offered for the result set is what facilitates our decision to develop two APIs instead of a single one.

The Local Result Set API views the result set as being local to the reactive kernel. In this case, operations on the result set can conceptually be considered to be instantaneous, thus satisfying the synchrony hypothesis [20]. This API will therefore make heavy use of user-defined external functions and procedures, which require synchrony. In practice, the Local Result Set API is for use when there is either a high-speed link between the database server and the reactive client, or when both reside on the same physical machine.

The Remote Result Set API considers the result set to be stored remotely to the reactive client, that is, the result set—in addition to the database—is viewed as part of the reactive system’s environment. This allows for the result set to be transferred row-by-row to the client, as is standard practice when accessing relational databases and closely fits with the provisions of MySQL. This is for use when, for example, the link between the database result set and the client is much slower than the duration of a reactive cycle. As a consequence, transfers are not instantaneous, and the Remote Result Set API cannot employ Esterel’s elegant mechanism of external functions and procedures, but must rely on signals and tasks instead. Indeed, a task concept has been incorporated into Esterel exactly for the purpose of handling external, asynchronous computations.

In the following, we discuss the design and usage of both APIs, first the Local Result Set API and then the Remote Result Set API. Some of the implementation details of the APIs will be presented later in Section 4; in particular, users will be able to customise the APIs according to their wishes, for example, in terms of the number of database connections required and the frequency with which they are accessed.

### 3.1. Local result set API

The easiest way to view the interaction between a reactive kernel programmed in Esterel and a database is to regard the database simply as an extension of the reactive kernel’s environment. For this reason all interactions with the database from within Esterel are modelled using input and output signals, as these are Esterel’s facilities for communicating with the environment. Therefore, to perform an operation on the database, a dedicated output signal is emitted, parameterised in a string that formulates a query in SQL syntax. The database’s response is awaited via a dedicated input signal whose parameter carries an identifier that points to the result set. During the time between the emitted query and the results returning, the database is queried and the whole result set is transferred back to the site that also runs the reactive kernel. Note that multiple databases can simply be supported by declaring a dedicated output and input signal for each database.

Once a database has been queried and a result set returned, data can be extracted from the result set via dedicated operations, for which we employ Esterel’s external function and external procedure facilities. This is possible since both the result set and the reactive kernel reside in the same memory, which implies that accesses of the result set by the kernel may be considered as instantaneous. If the query’s SQL command is one that does not return results, such as the command for the deletion of data items, then the only operation provided is one to check the number of affected rows. If the SQL command did return a result set, however, the set may be accessed by successively reading it row-by-row, extracting the specific data items from each row and coercing them into native Esterel data types. Once all rows have been processed, an operation will be called to free the memory occupied by the result set. Note that the three main operations (querying, retrieving results, and clearing results) must always be conducted in this order; for example, emitting a second query, before having received and cleared the results for the first one, results in undefined behaviour. This is also true analogously for the Remote Result Set API. Esterel’s interaction with a remote database and its local processing of result sets thus leads to the API displayed in Table 1. In the remainder of this section, we explain and illustrate the API’s services in more detail.

We begin with the formation of a query string containing SQL commands. Since Esterel does not provide any facilities for building strings, the string must be generated using a series of append operations. The API offers an append operation for each of Esterel’s native data types and implements these operations in Esterel using external procedures. As an example, the Esterel program fragment generates a query related to our warehouse case study, which uses an integer variable order_id:

```e
var query_str : string in
query_str := "select * from orders
where order_id = ";
call appint(query_str, order_id);
end var
```

As mentioned before, one interacts with the database via an output query signal and an input result signal. For each database used, these signals should be declared as such:

```e
output item_db_query : string;
input item_db_results : MYSQL_RES_ptr;
```

Each pair of signal names can be selected by the user. The mapping between these signal names and the actual databases is defined elsewhere and will be explained in Section 4. The data returned on a result signal is simply an identifier of the external type MYSQL_RES_ptr which is defined in our API’s implementation. Our framework effectively allows only one result set per database connection; however, if more result sets are required simultaneously...
within some Esterel application, additional connections to the same database may be declared.

In order to ensure that the results are received correctly from the database, the result signal should be awaited right after the emission of the query:

```plaintext
etitem_db_query("select * from item");
await item_db_results;
```

Note that replacing “await” by “immediate await” here would not change the semantics, since a query and the signal being returned is guaranteed to take at least one clock cycle. If two queries are issued simultaneously, then this can only be done via separate connections to the same database, that is, “pipelining” of queries is not supported by our API.

Moreover, the result signals of two simultaneous queries via two different database connections must be awaited in parallel or using immediate await statements in Esterel, as shown in the example below. This is due to the return signal from a database query only being present for one cycle; the program must register the signal on the cycle it is present or it will be lost. Therefore, if the results of two queries are being awaited, the program must be able to recognise both on the same cycle. For example, suppose we have a second database connection to the one shown above, with signal names order_db_query for the query signal and order_db_results for the signal returning the results. If we require a query on both connections to be issued simultaneously, then this should be done as follows:

```plaintext
etitem_db_query("select * from item");
etorder_db_query("select * from order");
await item_db_results || await order_db_results;
or:
```

```plaintext
etitem_db_query("select * from item");
etorder_db_query("select * from order");
await item_db_results;
await immediate order_db_results;
```

To check the success of an SQL command, the Boolean function check_result should be called and passed the identifier of the result set, that is, the value of the input result signal. If it returns true, then the query has succeeded and the operations described below may be used to access the data inside the result set. If it returns false, then the data in the result set is not valid. When querying a database, the query may fail in a number of ways, ranging from a timeout to an incorrectly formed SQL statement. The check_result function is suitably abstract so as to account for these problems, and simply lets the Esterel system know
the outcome of performing the query. It is the responsibility of the programmer to check the success of a query, and any failure modes should be specified in Esterel. An example of how this may be done using the `trap` and `exit` statements of Esterel’s error handling mechanism is given in the case study in Section 5.

For working with a result set that contains data—as opposed to an empty one returned by, for example, an SQL insert statement—rows must be declared inside Esterel. Rows are declared to be of external type `MYSQL_ROW` that is defined in our API’s implementation. The lifetime of any data loaded into a row from a result set lasts only as long as the result set itself, that is, up to the time the `clear_results` operation is called. Therefore, rows should only be declared locally and in such a way that their scope finishes before the call to `clear_results` occurs.

The functions provided to operate on the result set and rows will now be described. Most of the operations mirror the equivalent MySQL functions from the MySQL C API [14] on which our implementation is based (cf. Section 3.4). This is because the MySQL C functions are widely known. Moreover, at a later date, additional functions can easily be included, if desired. Function `get_next_row` is required to load data into a row from a result set. It is passed a result set identifier and, each time it is called, it will return the next row in the result set. Generally, the program will need to know how many rows there are in the result set and, therefore, how many times to call `get_next_row`. This is accomplished by a call to function `num_rows` which, when passed a result set identifier, returns the number of rows in the result set. Once a row has been loaded from a result set, data can be extracted using a `get<type>` function which is provided for each of the native Esterel data types. In addition to a row, an integer is also passed and indicates the index of the column from where the data is to be retrieved. The following is a simple example of data extraction using our API:

```c
var row_holder : MYSQL_ROW,
    item_name : string,
    item_location : integer in
row_holder := get_next_row(?item_db_results);
    item_name := getstr(row_holder, 1);
    item_location := getint(row_holder, 2);
end var;
```

In this case, the results are identified by the valued signal `item_db_results`, and the item’s name and location are stored in the second and third column of a row, respectively. Note that the indexing of columns is adopted from the C language and thus starts with 0.

Function `num_affected_rows` for accessing the result set is used when the result set contains no data but a user wants to know how many rows were affected by the SQL command. As such, `num_affected_rows` can be employed to test the success of an SQL query, for example, to check whether a delete query has had the desired outcome.

The final operation provided by the API, to which we have already referred above, clears the memory occupied by the result set:

```c
procedure clear_results():(MYSQL_RES_ptr);
call clear_results()(?item_db_results);
```

It is essential that there are no rows loaded from the result set after it is cleared, since the data within these rows is cleared with the result set as well.

### 3.2. Remote result set API

The Remote Result Set API should be used in situations where it is not feasible to transfer the entire result set to the system running the reactive kernel, that is, when both the database and the result set must be viewed as part of the environment. Since remote communication must be taken into account, the API is different to the Local Result Set API. This is because external functions and procedures can only be used in Esterel if their operations may be considered instantaneous [1]. Consequently, one must either employ Esterel’s task concept or must solely rely on signals for the Remote Result Set API. In both cases and as a consequence of operations on the result set not being instantaneous, the Esterel kernel must be informed when an operation is complete. This is accomplished by awaiting an “acknowledge” task completion signal after every operation. In the remainder we focus on our solution via tasks rather than signals, since this is the most elegant method for representing asynchronous interaction in Esterel. The interested reader is referred to [21] for an exposition of the solution employing signals.

An important aim of the Remote Result Set API is to minimise the amount of data that must be transferred, due to memory or bandwidth being at a premium. For this reason, rows are transferred to the system running the reactive kernel one at a time. There should be no overhead in returning the whole row, as opposed to the individual elements, since the row structure is specified in the query, and therefore it is the programmer’s decision exactly of what data the row consists. To prevent the complexity of handling multiple rows in the kernel, each database connection is limited to passing only one row at a time. This is a reasonable restriction since systems that use this API are unlikely to be performing complex database manipulations that require multiple rows.

Our API for remote result set access is displayed in Table 2. The remainder of this section explains the API’s services. Similar to the naming of the query and result signals in the Local Result Set API, each task `task_name` and function `function_name` is prefixed with a textual database identifier `db_id`, which we denote by `<db_id>_task_name` and `<db_id>_function_name`, respectively. Due to the restrictions imposed on tasks by Esterel, for each occurrence of starting a task in the syntax, a unique return signal is required to inform the reactive kernel of task completion. We represent this in the API by appending each return signal name, for example, `<db_id>_query_complete`, with
a unique number. For implementation reasons, this should start at 0 and increment by 1 for each task completion signal required:

```c
task <db_id>_perform_query() (string);
return <db_id>_perform_query_complete0(boolean);
return <db_id>_perform_query_complete1(boolean);
return <db_id>_perform_query_complete2(boolean);
...
```

Note that the offered string generation functions are identical to those in the Local Result Set API, as described in Section 3.1.

The main difference to the Local Result Set API is the way in which the results to a query are accessed. There is now one result set and one row per database defined, and since each signal is prefixed by a unique string, there is no need for a result set identifier to be returned. The only data returned after issuing a query is the success of that query. Therefore, after a query has been issued, the Boolean return signal for that task must be awaited syntactically just after starting the task:

```c
exec <db_id>_perform_query("select * from item")
   return <db_id>_perform_query_complete0;
await <db_id>_perform_query_complete0;
```

Note that the signal names `<db_id>_perform_query_complete0` in the return and await statements must match; they are a pair. It is an obligation on the programmer to ensure that there is no mismatch. This particularity cannot be resolved within the API as it is limited by Esterel's restrictions on task programming.

The value carried by the return signal is the same as that returned by the `check_result` operation in the Local Result Set API, that is, it should then be tested to determine whether the query has succeeded or not. If the query has succeeded and the result set is not empty, then the first row can be transferred by executing the `<db_id>_get_row` task. The Boolean return signal carries the value `true` if there exists a valid row to load, and `false` if there are no more rows available. Now that a row has been loaded, its elements can be accessed in a manner identical to that of the Local Result Set API, except that the database prefix `<db_id>` is used instead, since no row identifiers exist in this Remote Result Set API. The operations for determining the number of affected rows and the number of rows in the result set are not supported by MySQL when the result set is stored server side. Therefore, to step through the rows in a result set, the value of the return signal from the `get_row` task must be used as the loop variable. When there are no more rows left, it will carry value `false`.

### 3.3. Trade-offs between the APIs

While we have already pointed out the different situations for which the Local and Remote Result Set APIs have been designed, this section discusses the trade-offs between the APIs.

The Local Result Set API is based around flexibility and ease of use, which are a consequence of the fact that it is built upon external functions. In contrast, the Remote Result Set API is not as elegant by comparison, as it requires, for example, an “operation complete” signal to indicate the success of a database operation, whereas in the Local Result Set API this is given as the return value of an external function. Furthermore, the Esterel language specifies that, for each syntactic occurrence of starting a task, its completion must be awaited by a unique signal. Therefore, if an Esterel program includes a large number of occurrences of querying a particular database, then the number of return or task completion signals will be equally large. It should be noted that the querying of the database in the Local Result Set API is asynchronous, just as in the Remote Result Set API. However, in the former we do not use tasks to represent this, but instead rely on signals to illustrate the differences between the two methods and to emphasise that the database is considered part of the environment. As is the case regarding ease of use, the Local Result Set API also compares favourably to the Remote Result Set API regarding performance; this is because of the use of external functions rather than tasks.

Another difference between the two APIs is the number of rows that can be used simultaneously. The Local Result Set API supports as many rows as can be stored in memory, whereas the Remote Result Set API is limited to accessing one row at a time. However, since the Remote Result Set API is meant for use on small embedded systems with slow communication links, it is not expected that database operations requiring many rows will be commonplace.

Both APIs support connections to multiple databases. In the case of the Remote Result Set API, this can be used to overcome the limitation of one row per database by simply defining a second connection to the same database. Since database connections are generally permanent throughout the time a reactive system is running, our APIs provide no explicit facilities for connecting and disconnecting from a database. Instead, connection and disconnection is handled implicitly by the implementation of the APIs (see Section 4).

Finally, we discuss the issue of timeouts for when database transactions overrun. Timeouts must be present on all database transactions or the Esterel system could be left waiting for a completion signal that will never arrive. Currently, timeouts are simply coded within our APIs and assumes that each database transaction uses the same timeout value. If variable timeout values should be desired, one could adapt our APIs such that these timeouts could be encoded directly in Esterel; we leave this for future work. However, if a real-time database [22] with guaranteed response times is available, then timeouts would no longer be an issue. In certain application scenarios, it may even be possible that a real-time database could offer such good response times that the database would no longer need to be considered as part of the environment and could be directly integrated into the reactive system allowing the use of external functions and procedures for querying.
3.4. Comparison to other database APIs

This section compares our APIs to other interfaces to relational database management systems (RDBMS). Non-relational key/data-pair databases, such as Oracle Berkeley DB (http://www.oracle.com/technology/products/berkeley-db/), will not be considered here.

Our choice of MySQL as the underlying database roughly defines how our interfaces should be constructed. Although we chose to use the MySQL C API to access a MySQL database, we could also have employed the Open Database Connectivity (ODBC) interface for MySQL. However, this would limit the ways in which we could access the database; specifically, the ability to specify the manner in which the result set is transferred to the reactive kernel. Passing the whole result set back to the client is the standard method of returning a result set and, as such, is supported by the MySQL C and the ODBC interfaces. However, by directly building our interfaces on the MySQL C API, we may make use of mysql_use_result() which allows the row-by-row transfer of the result set and in turn the implementation of the Remote Result Set API.

Looking at the whole range of MySQL language interfaces, we find that the C interface exhibits the most functionality. As mentioned earlier, the ODBC interface is necessarily generalised since it must be able to interface to many different databases and hence does not have as much specific MySQL functionality as the native C interface. Scripting languages like PHP (http://www.php.net/) use a subset of the C interface and are on the whole simplified to correspond to the kind of rapid development web applications they are designed to support; they do not feature the row-by-row returning of the result set. One exception is the Perl interface (http://www.perl.org/) which provides a wrapper for using the MySQL C interface in Perl and thus supports the full functionality of the C interface.

Moving beyond MySQL, all the languages covered so far support the ODBC interface and, therefore, provide a homogeneous view of different RDBMS at the expense of some performance and native database functionality. Most RDBMS today support ODBC, so this is usually the database interface implementation of choice when database independence or access to different data sources is required. Some examples of other RDBMS's that support the ODBC are Oracle (http://www.oracle.com/), PostgreSQL (http://www.postgresql.org/), and SQL Server (http://www.microsoft.com/sql/). Java uses the JDBC (java.sun.com/javase/technologies/database) which is very similar to the ODBC; the JDBC is defined in Java while the ODBC provides database connectivity using native code. In both JDBC and ODBC, the underlying mechanisms that transfer the result set are hidden from the user in a driver layer. Thus, it is difficult to know how a result set is transferred; in fact, the method of transfer could vary in different ODBC drivers. Therefore, while it is clear that transferring the whole result set at a time is supported, it is not clear whether the row-by-row mechanism is supported.

4. IMPLEMENTATION OF THE APIs

This section gives some details on our implementation of the Local and Remote Result Set APIs; the full source code is freely available for download from [23].

Both APIs are implemented in a combination of Perl and C and rely on the MySQL C API [14]. The involvement of the Perl scripting language [24] in the realisation of the APIs may be surprising at first. The reason is our desire to support multiple databases with user-declared signal names.

Table 2: Services offered by the Remote Result Set API.
for emitting SQL queries and awaiting result sets. As reactive systems typically interact with an arbitrary but fixed number of databases, it is unnecessary to provide API services for dynamically binding signal names to databases. Even a static binding should not be defined within an Esterel program at all, as it is not part of specifying reactive behaviour. Instead, we choose to provide such a binding as a parameter to our Perl script for each API, which appropriately combines the C code generated by the Esterel compiler [12, 13] with C code implementing the API services used in the underlying Esterel program. Perl is an ideal choice to build this combination of C code and API services due to its fast and simple operators for writing and reading to text files. Furthermore, Perl’s regular expression feature is very useful for locating code segments that must be copied out of the Esterel generated C code in a robust manner.

The implementation of the Local Result Set API will be given preference here. This is because it is similar in spirit to the Remote Result Set API implementation, but does not require the added complication of threads and deeper knowledge of POSIX [25] as outlined in Section 4.2. The structure of an Esterel program once it has been translated into C, is that of an automaton which should be called on each instant of the system. Prior to being called, the input signals should be set up. The automaton will in turn call any output signals that are emitted on that instant. The implementation of the Local Result Set API is therefore based around the central automaton call. When a query is sent to the database, the automaton calls a user-written procedure that records its presence. Similarly, before the automaton is called, input signals are set up to reflect any response from the database.

4.1. Detailed implementation of the local result set API

The Local Result Set API script is used as follows:

```
gendb.pl <Main module Name>
<Max length of strings and queries>
<DB Name>
<Host>
<User>
<Password>
<Signal name for emitting query>
<Signal name for returning results>
```

The parameters carry the following meanings.

(i) **Main Module Name.** The name of the module compiled with Esterel; it is assumed that the autogenerated C code is in `<Main Module Name>.c`.

(ii) **Max length of strings and queries.** Since all strings must have a fixed-length representation in Esterel, this specifies what the maximum length is. It must be ensured that the program never exceeds it. This includes getting strings from the database using `getstr()`. The type of data that `getstr()` is called on should be guaranteed not to exceed the maximum length; for example, if the maximum length was 200, the database field could be of type VARCHAR[150], and the string could be appended by up to 50 characters by the user to represent the SQL query.

(iii) **DB Name.** The name of the database on the host.

(iv) **Host.** The location of the MySQL DB, if hosted locally, then localhost is to be used.

(v) **User.** The username to be used to connect to the database.

(vi) **Pass.** The associated password.

(vii) **Signal name for emitting query.** The name of the signal that is used to send a query to this database, that is, the name of the signal listed as:

```
output <Signal name for emitting query> : string;
```

for this signal in Table 1.

(viii) **Signal name for returning results.** The name of the signal that will be awaited for results, that is, the name of the signal defined as:

```
input <Signal name for returning results> : MYSQL_RES_ptr;
```

for this signal in Table 1.

The last six parameters may be repeated any number of times for additional databases, whence a single database can be accessed via multiple connections.

Note that we have intentionally not provided default values for the parameters of the gendb script. This is because we believe that every application scenario will be different and that there are no sensible default values.

After the arguments to the script have been processed, it continues by copying the relevant parts of the Esterel generated C code into the new C code file. For example, the following Perl code copies consecutive lines of EST_FILE to the file OUT until the line `#include "$main_module.h"` is found. During this process, when the line defining the maximum string length is found, it is replaced with the user-defined string length (see Table 3).

Perl’s treatment of file handles and regular expressions makes this kind of code manipulation very easy. Note the command `$line =<EST_FILE>` which reads the next line of EST_FILE into the variable `$line`, and the regular expression operator `=~` which tests if a string variable contains a regular expression. If the symbol `\` occurs inside a sting it must be escaped using `\`. Furthermore, variables can be directly included in strings. This occurs in the string `"#include \"$main_module.h\" \n"`, where the string value in the variable `$main_module` is substituted wherever `$main_module` appears.

The next operation is to define all MySQL variables; this is based on the number of database connections required by the user. In the code below, the array `dbs[$x]` stores all
of the user-entered information for the database connection $x$. The entry $\text{dbs}[x][0]$ is the id for that connection (see Table 4).

After this, the functions and procedures defined in Table 1 are produced. For example, the Esterel function $\text{get\_next\_row(}\text{MYSQL\_RES} \text{ptr}) : \text{MYSQL\_ROW}$; is produced like this: $\text{MYSQL\_ROW get\_next\_row(}\text{MYSQL\_RES} \text{ptr res}) \{ \text{return mysql\_fetch\_row(res);} \}$

Next, the signal output function is produced for each database query signal. This function is responsible for submitting the query to the database, and for retrieving the result or reporting a failure. The code is too long to include here, but the interested reader is referred to view it in the source code of the Perl script where the C functions used to interface with MySQL are clearly seen [23]. Finally, the main C procedure is produced. It commences by initialising all database variables and the automaton state. If a connection to a required database cannot be opened, execution is terminated. The main reactive loop is started, inputs are set up, and the main automaton step procedure is called.

Section 3 mentions that databases are running asynchronously to reactive kernels, since database transactions are relatively complex and thus cannot be assumed to respect the synchrony hypothesis. The current implementations of our Local Result Set API does not implement this intention explicitly. This is because the output procedure called by the reactive kernel to process a database interaction should be able to be considered instantaneous. However, due to the fact that we wait for the results from the database in the same procedure, this will not be the case. This is however not a problem in the API since, immediately following a query emission, the result input signal should be awaited. Since there can be no statement between these two operations, it does not matter that the reactive cycle has temporally paused. Timeouts are used to make sure the database query output procedure cannot execute forever, in case of a problem with the connection to the database. Note that this behaviour is specified at the API-level rather than in Esterel. The Esterel programmer can test for timeouts via the check_result function and handle them as desired.

However, if an implementation of our API should explicitly support the asynchronous view between synchronous reactive kernels and databases, then it would not be difficult to do so by introducing threading using POSIX [25]. The query output function would need to be changed to inform a separate thread that would perform the database operation at hand. At the start of each iteration of the reactive system, the thread would then be queried to determine if any database operations have finished. If so, the location of the result set would be passed to the reactive kernel thread, and the database result input signal would be emitted in that cycle.

### Table 3

<table>
<thead>
<tr>
<th>Line 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ((x = &lt;EST_FILE&gt; ne &quot;#include &quot;${main_module}.h&quot; \n&quot;) {</td>
</tr>
<tr>
<td>if (x = ~/#define STRLEN/)</td>
</tr>
<tr>
<td>print OUT (&quot;#define STRLEN ${max_strlen}\n&quot;);</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>print OUT $x;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>Line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (my $x=0; $x &lt; $num_dbs; $x++)</td>
</tr>
<tr>
<td>print OUT ('\n/Global variables for ${dbs[$x][0]} \n''n&quot;);</td>
</tr>
<tr>
<td>print OUT ('MYSQL init,${dbs[$x][0]},+sock,${dbs[$x][0]};\n''n&quot;);</td>
</tr>
<tr>
<td>print OUT ('MYSQL_RES $result,${dbs[$x][0]};\n''n&quot;);</td>
</tr>
<tr>
<td>print OUT ('int query_pending,${dbs[$x][0]},</td>
</tr>
<tr>
<td>query_succeeded,${dbs[$x][0]};\n''n&quot;);</td>
</tr>
</tbody>
</table>

4.2. Discussion of the implementation of the remote result set API

In the implementation of the Remote Result Set API, the actual database transactions are handled in a separate thread to ensure that they do not interfere with the periodic nature of the automaton. One thread is made for each database connection. Its task is to regularly check if the automaton has requested it to perform a database operation and, if so, it accesses the associated parameters and performs the operation. When the operation is complete, its results are reported back through a shared-data store and it waits for the next database operation.
The passing of information between the thread running the main automaton and a thread running a database connection must be carefully controlled in order to ensure the automaton thread does not become stalled. Broadly speaking, this means imposing restrictions on when data can be sent to a database thread. In terms of our API, this equates to the restriction that only one database operation can be performed at a time, that is, the operation must be fully complete before the next is begun. To explain the interaction between the two types of threads, we will now give a code skeleton of each thread, followed by an example of how a typical database operation is accomplished (see Table 5).

A typical database operation will now be described to illustrate where blocking can occur and how it is handled. Let us begin by assuming that we are currently executing the automaton step procedure (a) and that a task requiring a database operation has just been started. When the automaton step procedure ends, we will execute the output function (b) associated with the task that was just started. This function will wait until it has acquired a lock on Mutex1. Once the lock is acquired, it will record that the database operation thread (c) must perform an operation when it is next scheduled. It then releases the lock and returns.

This is the only part of the process where blocking can occur as it is possible that the database connection thread has the lock on Mutex1. However, since our API only allows one operation to be executed at a time, the database connection thread can only be checking whether there is an operation to execute. This check is very fast, and the lock will be released and its runtime yielded quickly. This will allow the output function (b) to acquire the lock and complete.

The next time the database connection thread is scheduled after the output function has completed, it will try to acquire the lock on Mutex1 and find that there is an operation to execute. It will then execute this operation, record the results, release the lock and yield its runtime. (Since there can only be one database operation running at a time, it does not matter whether the shared data source or the thread encapsulating the database query is locked.) The next time the setup inputs phase (d) of the main reactive loop is executed (i.e., after the database connection thread has finished its operation), it will acquire the lock on Mutex1 and setup the automaton inputs with the results of the database operation. After this, it will release the lock on Mutex1, and the API’s side of the operation is complete.

If the acquisition of the lock fails during the input setup phase (d), then the database operation thread is busy and no results need to be passed back to the automaton; therefore, the input setup phase for database results can be skipped in this iteration. In case the database connection is unreliable, Mutex1 could become locked forever; timeouts are included to prevent this. The full implementation details of the Remote Result Set API can be found in [23].

5. CASE STUDY

In this section we present a case study demonstrating the utility of our APIs; an automated warehouse storage system modelling a direct order company, where items from orders are picked, stored and finally removed from the warehouse. This involves producing a warehouse containing various items and a robot capable of moving the items within the warehouse, for which we use Lego Mindstorms robotics kits [17] (cf. Figure 1). Lego Mindstorms provides both a construction tool with sensors and actuators and a microcontroller, called the RCX, which is capable of running a reactive kernel programmed in Esterel. The database behind our warehouse model is that of a standard order system but which also includes mapping data about the location of the items. As this case study is meant to exemplify the use of our database APIs, only the part of the solution employing the APIs is emphasised below. All code and associated scripts for the case study may be downloaded from [23], where the interested reader can also see a video of our warehouse system in operation.

5.1. Lego Mindstorms, the RCX and BrickOS

Lego Mindstorms is a platform for building computer-controlled robots using the Lego system [17]. At the heart of Lego Mindstorms is the RCX. This “brick” is a small battery-powered computer which is capable of controlling up to three actuators and reading up to three sensors. In Lego, actuators are normally motors, and sensors can be light, rotation and touch sensors. Each RCX also provides an infrared transmitter and receiver used for both downloading programs from a PC and for inter-RCX communication. The infrared download device used on the PC can also participate in communications with RCXs.

The RCX provides great flexibility through its reprogrammable firmware. BrickOS [26], formerly known as LegOS, is an open-source replacement firmware for the RCX. It boasts a number of features that make it considerably more powerful than the standard Lego firmware. Foremost, it allows programs written in the C language to be executed on the RCX. Obviously, this is especially important for this
### Table 5

<table>
<thead>
<tr>
<th>Thread: Database Connection 1 (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>if (acquire lock on Mutex1)</td>
</tr>
<tr>
<td>if (database operation to perform)</td>
</tr>
<tr>
<td>Perform request;</td>
</tr>
<tr>
<td>Record results;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Release lock on Mutex1;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Yield runtime;</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread: Esterel Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>// Setup inputs (d)</td>
</tr>
<tr>
<td>if (acquire lock on Mutex1)</td>
</tr>
<tr>
<td>if (database operation to perform)</td>
</tr>
<tr>
<td>if (database operation completed)</td>
</tr>
<tr>
<td>Setup up the automaton inputs with the database operation result;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Release lock on Mutex1;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>Perform Main Automaton Call; (a)</td>
</tr>
<tr>
<td>// Process outputs</td>
</tr>
<tr>
<td>For any new task that was started in the last cycle</td>
</tr>
<tr>
<td>Call the output function associated with that task (e.g., Start Task 1);</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function: Start Task 1 (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait until (acquire lock on Mutex1)</td>
</tr>
<tr>
<td>Record that THREAD: Database connection 1 must perform a db operation;</td>
</tr>
<tr>
<td>Release lock on Mutex1;</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

---

project since the Esterel compiler [12] generates C code as target language. BrickOS also provides infrared communication through the Lego Network Protocol (LNP) [26] which allows message broadcast and directed transmissions.

### 5.2. Hardware

The hardware requirements of our warehouse storage system are high in Lego Mindstorms’ terms, requiring more sensors and actuators than one RCX can control. Therefore, it is necessary to use two RCXs, one to control the movement of the robot and the second to control the employed forklift installed on top of the movement unit, hereafter referred to as the movement RCX and Forklift RCX, respectively. Again, communication between the two RCXs is handled using the infrared link provided on each RCX. Because the Forklift RCX does not need to communicate with the PC running the database system, the infrared download tower is set up to allow the following communication to take place: PC to movement RCX and movement RCX to forklift RCX, as shown in Figure 2.

### 5.3. Software

The software for the warehouse system is structured in three layers: database access layer, route interpretation layer, and hardware layer. Each device used in the system, the two RCXs and the PC, executes a reactive kernel programmed...
in Esterel; note that these are running asynchronously to each other, due to the nonnegligible delay caused by infrared communication channels. The database layer runs on the PC and is responsible for accessing the warehouse's database and for generating routes through the warehouse for the robot to execute. The second layer runs on the movement RCX and interprets the route sent from the PC. The third layer is responsible for interacting with the Lego hardware and performs operations such as move the robot and pick up the item. This layer is present on the movement RCX and the forklift RCX. The signals used for communicating between the various layers are shown in Figure 2.

5.3.1. Database

Our warehouse’s database models a simple ordering system. Since the main emphasis within this case study is on retrieving spatial data, the aspects of the database concerning ordering information are kept as simple as possible; a customer may make multiple orders, each order is identified by an order id and must contain one or more order lines, and each order line refers to exactly one item. The corresponding database schema is given in Figure 3.

Stored separately is a table describing the drop-off bins in the warehouse. A drop-off bin is used by the robot to place parts of an order before it is complete. When the order is complete, the items that are contained in the bin are removed, and the bin is then ready for the next order. For each bin, its location is stored, as is the id of the order in the bin, in case the bin is in use.

5.3.2. Database access layer

The database access layer uses our Local Result Set API. We chose the Local Result Set API for our case study since the Esterel program interacting with the database will be running on the same machine as the MySQL server. Therefore, result sets will be stored in the same memory as the Esterel program is run in, and can be considered to be accessible instantaneously by the reactive kernel. Two connections to the same database are maintained since, at one point in the program, it is necessary to manipulate the database while retaining the result set of an earlier operation. The input and output signals for the main connection to the database are called orders_query_out and orders_results, respectively, and for the additional connection stock_query_out and stock_results, respectively, since those are only used to update stock levels:

```plaintext
output orders_query_out : string;
input orders_results : MYSQL_RES_ptr

output stock_query_out : string
input stock_results : MYSQL_RES_ptr;
```

Since the database access layer's only function is to wait for an order that needs to be picked and then to instruct the robot how to pick it, the main module is constructed as a loop. Inside the loop is a trap statement which handles all the ways in which the database and the system can fail (see below). Therefore, all failure modes are dealt with in one place, and the error handling process is simplified.

The operation of the database access layer is roughly as follows. First, an order is retrieved from the database. One of the lines of this order is then extracted and the item details are stored locally. The robot is then sent to retrieve all items, one by one, and to deliver them in an available drop-off bin using a pregenerated route through the warehouse. After each item has been collected, it is necessary to update the stock level of the item's type. However, at this point the database result set still contains uncollected order lines which are required for later in the program's
execution. Therefore, the second database connection is used to perform the update without overwriting the result set containing the order’s details.

Since the functionality of the database access layer is quite simple and most database operations are effectively the same, only two database operations will be discussed. The retrieval operation starts by emitting the query on output signal orders_query_out and then awaits the results (see Table 6).

The returned results are then checked for validity using our API function check_result. If the query has succeeded, then num_rows is called on the result set to see if any rows were returned. If rows have been returned, then there are orders waiting and, consequently, the first row is loaded into the local variable row. The details of the row are then retrieved using the get <type> functions and emitted on the corresponding local signals: order_id, customer_id, customer_name, and customer_address. Now that the result set is no longer needed, the clear_results procedure is called. The following code captures these steps:

```cpp
if (check_result(orders_results)) then
    var row : MYSQL_ROW in
        row := get_next_row(orders_results);
        emit order_id( getint(row,0) );
        emit customer_id( getint(row,1) );
        emit customer_name( getstr(row,2) );
        emit customer_address( getstr(row,3) );
    end var
    call clear_results(orders_results);
else ...
```

Note that we only look at the first order and then clear the result since we simply wish to process one order at a time. Once the order_id has been obtained, we are not interested in the result set any more.

There are two ways in which this operation for retrieving orders can fail: firstly, if the query fails and, secondly, if there are no waiting orders. Each way is catered for by an exit statement which corresponds to the trap mentioned above:

```cpp
if (check_result(orders_results)) then
    if (num_rows(orders_results) > 0) then
        %Code snipped
    else
        call clear_results(orders_results);
        exit no_waiting_orders;
    end if;
else
    exit bad_query;
end if;
```

In each case, the program flow jumps to the end of the loop and emits an appropriate error message before pausing and then repeating the main loop. Note that function clear_results must be called after it has been determined that there are no waiting orders, freeing the memory occupied by the result set.

The program then continues according to the pseudocode shown in Table 7. Generally speaking, the items in an order are retrieved next. For each item, its location is looked up and a route to retrieve that item is generated and then executed by the robot. After the last item is stored, the process is repeated for the next order. The database is updated continually to reflect the current state of the warehouse (stock levels, etc.) and orders (if an order is being picked or completed and stored in a drop-off bin). For example, the code in Table 8 details how a stock update is performed.

The generation of the pick-up and drop-off routes is the only non-database-related function of the database access layer. A function generate_route_to, which is not displayed here but can be found in [23], generates a string consisting of op codes that represent the operations the robot

---

**Table 6**

<table>
<thead>
<tr>
<th>customer</th>
<th>orders</th>
<th>order_lines</th>
<th>items</th>
<th>dropoff_points</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>customer_id</em></td>
<td><em>order_id</em></td>
<td><em>order_id</em></td>
<td><em>item_id</em></td>
<td><em>bin_id</em></td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>status</td>
<td>stock_level</td>
<td>name</td>
</tr>
<tr>
<td>address</td>
<td></td>
<td>dropoff_point</td>
<td>pos_x, pos_y</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** The employed database schema.
loop forever
  if (orders are waiting) then
    Store the details of one order;
    if (free drop-off bin is available) then
      Store bin location;
      Set order status = PICKING;
      Get order lines;
      while (there are still order lines left to process)
        Robot picks up item(s) and drops in bin;
        Update the number of items in bin;
        Update item stock level;
      end;
      Set order status = STORED;
      Update bin with order number stored in it;
    end
  end
  PAUSE
end

Table 7: Pseudocode describing the operation of the Esterel program running on the PC that accesses the database.

```plaintext
var query_str : string in
  query_str := "update item set stock_level = ";
  call appint(query_str, ?item_stock_level - ?item_quantity);
  call appstr(query_str, " where item_id = ");
  call appint(query_str, ?item_id);
  emit stock_query_out(query_str);
  await stock_results;
endvar;

if (check_result(?stock_results)) then
  if (num_affected_rows(?stock_results) < 1) then
    exit stock_update_failed;
  else
    exit bad_query;
  end if;
else
  call clear_results(?stock_results);
end if;
```

Table 8

must perform to pick up that item and to return to the communication point in the warehouse. The string is sent to the robot via a signal, and then another signal indicating the completion of executing the route is awaited. After the pick-up route is complete, a drop-off route is generated and executed in a similar fashion. By using external functions to generate the route, the warehouse can be redesigned in any way consistent with the item locations stored in the database, and only the two route generating functions will have to be rewritten.

5.3.3. Route interpretation layer

To interpret a route string of op codes, a number of external C functions are provided that extract parts of the string; details can again be found in [23]. First, a
function \texttt{get\_num\_ops()} is called to determine how many separate operations the route string contains. A loop is then repeated this number of times. On each iteration, a function \texttt{get\_op()} is invoked to extract the type of operation and \texttt{get\_param()} to extract the parameters to the operation. Once the operation type has been determined, an emission is made on the appropriate signal with the value obtained from \texttt{get\_param()}. When the robot has completed the operation, either the \texttt{movement\_op\_complete} or \texttt{forklift\_op\_complete} signal is emitted, informing the route interpretation layer that the robot is ready for the next operation. When all operations have been performed in this manner, the signal \texttt{route\_complete} is emitted, which lets the database access layer know that the robot has finished its route.

5.3.4. Hardware layer

The movement RCX is required to run both the route interpretation layer and part of the hardware layer. To accomplish this, both layers are run in parallel and local signals are used to communicate between them. To communicate with the hardware layer running on the forklift RCX, the output signals \texttt{pickup\_item} and \texttt{drop\_item} and the input signal \texttt{forklift\_op\_complete} are used (cf. Figure 2). These signals, combined with the signals of the hardware layer running on the movement RCX, that is, signals \texttt{move\_forward, turn\_left, turn\_right} and \texttt{movement\_op\_complete}, give the complete range of commands provided by the hardware layer.

6. CONCLUSIONS

This article presented two APIs for interfacing the synchronous programming language Esterel to the relational database MySQL. The \texttt{Local Result Set API} assumes the result set to a database query to be stored locally to a reactive Esterel kernel and largely relies on the external function concept of Esterel. The \texttt{Remote Result Set API} considers the result set to be stored remotely and is realised via Esterel’s task concept. To the best of our knowledge, this article provides the first detailed discussion of, as well as APIs for, accessing relational databases from a popular, industry-strength language which is used for programming embedded controllers.

Both of our database APIs worked well in testing. In particular, the \texttt{Local Result Set API} is extensively used in our warehouse case study and, although none of the database operations there are particularly complex, they are representative of the kind of database operations performed by embedded systems. In contrast to the elegance exhibited by the \texttt{Local Result Set API}, the \texttt{Remote Result Set API} appears to be slightly more complex. This is due to the modelling of all database operations as tasks, which became necessary since remoteness implies that one cannot expect instantaneous responses and, hence, cannot employ external functions for accessing result sets.

The development of the \texttt{Local Result Set API} also showed that the language extension features provided by Esterel are indeed sufficient for our application. Given the constraints imposed by a synchronous language, we feel that the developers of Esterel have selected an appropriate set of extensibility options that allow the language to be extended while keeping it within the synchronous paradigm. Of course, it is possible to abuse these features by allowing external functions and procedures to execute for a significant portion of time. However, if used correctly, the extensibility options allow great flexibility, from the ability to represent abstract data types to being able to pass these data types to external code.

It should be emphasised that the introduction of a database in an Esterel reactive system using either of our APIs does not undermine Esterel’s synchrony hypothesis. However, since the response times for returning query results or for accessing remote result sets cannot be guaranteed, the system can only wait for a signal that may never arrive. If the database is one that can provide guaranteed response times, such as a real-time database, the problem is elevated. Otherwise, the issue must be solved via timeouts in system design. In our case study, all database operations are performed at non-time-critical points, whence any unexpected delay from the database simply results in the system pausing, not malfunctioning.

Finally, it must be mentioned that our approach is not restricted to the particular database MySQL employed by us. Indeed, our APIs can easily be adapted to those databases that support the \texttt{Open Database Connectivity} (ODBC) API [27], since the ODBC operations are quite similar to those provided by MySQL.

ACKNOWLEDGMENTS

This research was supported by an Undergraduate Research Bursary of the Nuffield Foundation (Grant no. URB/01528/G). The authors would also like to thank the anonymous reviewers for their detailed suggestions, as well as the participants of the SYNCHRON’04 workshop at Schloss Dagstuhl, Germany, for their constructive comments on an early version of this work.

REFERENCES


Compositional design of systems on chip from preverified components helps to achieve shorter design cycles and time to market. However, the design process is affected by the issue of protocol mismatches, where two components fail to communicate with each other due to protocol differences. Convertibility verification, which involves the automatic generation of a converter to facilitate communication between two mismatched components, is a collection of techniques to address protocol mismatches. We present an approach to convertibility verification using module checking. We use Kripke structures to represent protocols and the temporal logic ACTL to describe desired system behavior. A tableau-based converter generation algorithm is presented which is shown to be sound and complete. We have developed a prototype implementation of the proposed algorithm and have used it to verify that it can handle many classical protocol mismatch problems along with SoC problems. The initial idea for ACTL-based convertibility verification was presented at SLA++P ’07 as presented in the work by Roopak Sinha et al. 2008.

Copyright © 2008 Roopak Sinha et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

Systems on chip (SoC) are complex embedded systems built using preverified components (called intellectual property blocks or IPs) chosen from available IP libraries. The various IPs of an SoC may be interconnected via a central system bus on a single chip.

The integration of IPs into an SoC involves addressing key compatibility and communication issues. One such important issue is that of protocol mismatches which arise when two or more IPs in an SoC have incompatible protocols. Protocol mismatches may prevent meaningful communication between IPs as they may not be able to correctly exchange control signals (due to control mismatches), exchange data (due to data-width mismatches), and/or connect to each other at all (due to interface mismatches). For example, two IPs that have different handshaking sequences have inherent control mismatches. If their word-sizes (the sizes of their data read/write operations) differ, they suffer from data-width mismatches. Interface mismatches occur when two IPs use different naming conventions for the same control/data signals, disallowing straightforward integration of the IPs into an SoC. Unless any mismatches between IPs are resolved, it is impossible to build an SoC that is consistent with the intended system-level behavior.

Protocol mismatches may be resolved if one or all of the mismatched IPs are modified. However, this process of manual modification is usually ineffective because firstly, it requires significant time and effort to modify complex IPs, and secondly, if requirements change later in the design cycle, further repetitions of manual modification might be required. Protocol mismatches may also be resolved by using convertibility verification (or protocol conversion) techniques, which involve the generation of a converter, some additional glue, that guides mismatched components to satisfy system-level behavioral requirements while bridging the mismatches as well.

A possible solution to convertibility verification comes from the verification of open systems using module checking [1]. An embedded system may behave differently under different environments, and verification of an embedded system under the influence of different environments was studied in [2]. In [3], local module checking, an approach to build an environment under which a system satisfies a given specification, was presented. In this paper, we adapt local module checking for convertibility verification to build a converter under which IPs satisfy given system-level specifications.

The main features of the proposed solution are as follows. Firstly, IPs are represented using Kripke structures...
and communicate synchronously with each other through input/output control signals. The desired behavior of the interaction between IPs is described using the temporal logic ACTL. ACTL is the universal fragment of CTL which allows only universal path quantification. ACTL is used for two main reasons. Firstly, the intended behavior of the interaction between mismatched protocols is usually required to be exhibited over all paths. Hence, universally quantified specifications are usually sufficient to describe such intended behavior. Secondly, the handling of existentially quantified formulas (EU and EG) results in the high (exponential) complexity of module checking [2].

Given two mismatched IPs and a set of ACTL specifications to be satisfied by their interaction, an automatic converter generation algorithm is employed to generate a converter if possible. We prove that the algorithm is sound and complete and can be used to resolve many commonly encountered control mismatches.

The rest of this paper is organized as follows. Section 2 discusses literature related to the proposed approach. An illustrative example is presented in Section 3. Section 4 presents the formal description of protocols and their interaction. Section 5 shows how specifications are described in our setting. Section 6 defines converters and shows how they control a given pair of mismatched protocols. The converter generation and extraction algorithm is presented in Section 7. Implementation results are given in Section 8 followed by concluding remarks in Section 9.

2. RELATED WORK

A number of techniques have been developed to address the problem of protocol conversion using a wide range of formal and informal settings with varying degrees of automation—projection approach [4], quotienting [5], conversion seeds [6], synchronization [7], and supervisory control theory [8], just to name a few. Some techniques, like converters for protocol gateways [9] and interworking networks [10], rely on ad hoc solutions. Some other approaches, like protocol conversion based on conversion seeds [6] and protocol projections [4], require significant user expertise and guidance. While this problem has been studied in a number of formal settings [4, 6, 7, 11], only recently some formal verification-based solutions have been proposed [8, 12–14].

The closest ones to our approach are [12, 13]. In [12], the authors present an approach using finite state machines to represent both the protocols and the desired specifications. A game-theoretic framework is used to generate a converter. This solution is restricted only to protocols with half-duplex communication between them. D’Silva et al. [13, 15] present synchronous protocol automata to allow formal protocol specification and matching, as well as converter synthesis. The matching criteria between protocols are based on whether events are blocking or nonblocking and no additional specifications can be used. The approach allows model checking only as an auxiliary verification step to ensure that the generated converter is correct.

In contrast to the above techniques, we use temporal logic to represent desired functionality of the combined protocols. Being based on temporal logic, our technique can define desired properties succinctly and with a higher level of granularity. For example, a desired behavior of the combination may be sequencing of events such that event $a$ in protocol $P_1$ always happens before event $b$ in $P_2$. Also, as our technique is based on the (tableau-based) module checking algorithm, the converter synthesized is correct by construction, unlike [13].

The presented approach, based on extending the local module checking algorithm [3], is similar to the synthesis of discrete controllers using temporal logic specifications [16, 17]. In [17], a controller synthesis paper using temporal logic (CTL) specifications, the authors note that the problems of module checking and supervisory control are duals of each other. The problem of conversion can therefore be handled by extending any of these dual approaches. However, it must be noted that the conversion problem addressed in this paper is not a straightforward implementation of either module checking or supervisory control. It requires nontrivial extensions including the handling of multiple protocols, input-output signals, exchange of signals between protocols, and special states such as output-only states or delayed-output states (described later in Section 4).

3. ILLUSTRATIVE EXAMPLE

We motivate our protocol conversion technique using the following example of two mismatched IPs. Figure 1 shows the protocols of two IPs, handshake and serial, that need to communicate with each other. This example was presented originally in [12] and has been adapted by adding state labels to each protocol. The handshake protocol emits the outputs req and gnt which can be read by the serial protocol. The IPs are expected to communicate by exchanging these I/O signals infinitely often.

The protocols of the two devices are described as follows. In its initial state $s_0$, the handshake protocol emits the signal req and makes a transition to state $s_1$. In $s_1$, it can wait for an indefinite number of clock ticks (shown as the self-loop on $s_1$) before writing the signal gnt and moving back to its initial state $s_0$. Outputs are distinguished from inputs by placing a bar over them.
The serial protocol operates as follows. In its initial state $t_0$, it waits indefinitely (using the self-loop on $t_0$) for the signal req and makes a transition to state $t_1$ when req is available. In $t_1$, it immediately requires the input gnt to make a transition back to its initial state $t_0$.

Intuitively, the mismatch between the two protocols can be described as follows. The handshake protocol can wait indefinitely before writing the signal $\overline{gnt}$ once it has emitted the signal req. On the other hand, the serial protocol needs gnt to be available immediately following the reception of req. Given this basic difference in their protocols, it is possible that their interaction results in the violation of system-level specifications, such as

1. a protocol must never attempt to read a signal before it is emitted by the other;
2. each emission of a signal by a protocol must be read successfully by the other protocol before it attempts to emit more instances of the same signal.

Due to different protocols, the IPs cannot guarantee satisfaction of the above specifications. To enable these IPs to interact in the desired fashion, a converter to bridge inconsistencies between their protocols is generated. The converter acts as a communication medium between the protocols, as shown in Figure 1, and guides the interaction between the two protocols by controlling the exchange of control signals between them. The resulting system, in the presence of the converter, guarantees the satisfaction of the above specifications. To enable these IPs to interact in the desired fashion, a converter to bridge inconsistencies between their protocols is generated. The converter acts as a communication medium between the protocols, as shown in Figure 1, and guides the interaction between the two protocols by controlling the exchange of control signals between them. The resulting system, in the presence of the converter, guarantees the satisfaction of the given system-level properties.

We use the above handshake-serial example throughout the rest of this paper to illustrate our approach.

4. MODEL OF PROTOCOLS

4.1. Kripke structures

Protocols are formally represented using Kripke structures, defined as follows.

**Definition 1** (Kripke structure). A Kripke structure (KS) is a finite state machine represented by a tuple $(AP, S, s_0, \Sigma, R, L, \text{clk})$, where

1. $AP$ is a set of atomic propositions,
2. $S$ is a finite set of states,
3. $s_0 \in S$ is the initial state,
4. $\Sigma = \Sigma_I \cup \Sigma_O \cup \{T\}$ is a finite set of events, where $\Sigma_I$ is the set of all input events, $\Sigma_O$ is the set of all output events, and $T$ is the tick event of the clock clk,
5. $R: S \times \Sigma \rightarrow S$ is a total (with respect to $S$) and deterministic transition function,
6. $L: S \rightarrow 2^{AP}$ is the state labelling function,
7. $\text{clk}$ is the system clock event.

Transitions of a Kripke structure can be divided into three categories: input transitions, output transitions, and tick transitions. All three types of transitions trigger with respect to the tick $T$ of clk which represents the rising edge of the clock. An input transition from a system’s current state triggers when the input trigger of the transition is present at the next tick of the clock. Similarly, an output transition triggers when the system, at the next clock tick, emits the output corresponding to a given transition. Finally, in case of a tick transition, a transition simply triggers on the next clock tick without the Kripke structure reading any inputs or emitting any outputs. A tick transition is used to implement delays. In all three cases, the Kripke structure moves from a current state to a destination state.

The presence of an event $a$ as an input is denoted as $a$ $(a \in \Sigma)$ over a transition whereas the emission of a signal $b$ as an output is denoted as $\overline{b}$ ($\overline{b} \in \Sigma$). In case no input/output triggers are present, the transition with respect to solely the clock’s tick event $T$ is taken $(T \in \Sigma)$. Note that $R$ is total with respect to $S$, implying that each reachable state in a KS must have at least one outgoing transition. Furthermore, $R$ is deterministic, implying that each reachable state can have only one transition to a unique successor state for any particular input or output event. The shorthand $s \xrightarrow{a} s'$ is used to represent the transitions of a Kripke structure $(s' = R(s, a))$.

**Restrictions**

The following restrictions are placed on Kripke structures representing IP protocols.

1. Well-formedness. A well-formed KS has only the following types of states (Figure 2 shows the different types of states in a well-formed Kripke structure).

   - **Input states.** A state $s \in S$ is an input state if none of its transitions results in the emission of an output. In other words, all of its transitions are triggered by input events or $T$ (see Figure 2(a)). Whenever the KS reaches $s$, if no input triggers are present in the next tick, the tick transition is taken. In case there is no tick transition, $s$ must be provided with an input that enables one of its transition.
   
   For example, consider the state $t_1$ of the serial protocol presented in Figure 1. The state must be provided with the input gnt in the next tick, otherwise its behavior is not defined.

   - **Output-only states.** A state $s \in S$ is an output-only state if it has only one transition that is triggered by an output
signal (see Figure 2(b)). Whenever a KS reaches an output-only state, in the next tick, the lone output transition is taken (and the corresponding output is emitted). We introduce the function OutputOnly, where OutputOnly(s) returns true when the state s is an output-only state.

(iii) Delayed-output states. A state s ∈ S is a delayed-output state when it has exactly two transitions; one triggered by an output signal and the other triggered by T. The tick transition must be a self-loop (s T → s) and must model an arbitrary delay before the output transition is taken (see Figure 2(c)). We introduce the function DelayedOutput, where DelayedOutput(s) returns true when the state s is a delayed-output state.

We restrict states in a well-formed KS to have at most one output transition to ensure determinism. A state with two output transitions is nondeterministic because whenever the KS reaches such a state, it is not known which output will be emitted (and which transition will be taken) in the next tick.

For example, state s0 of the handshake protocol presented in Figure 1 is a delayed-output state with a tick transition modelling an arbitrary delay. Hence, DelayedOutput (s0) = tt.

(2) Shared clock. Each KS must execute using the same clock, allowing multiple protocols to only make synchronous transitions. This restriction assumes that there are no clock mismatches between protocols.

Consider the protocol for the handshake IP presented in Figure 1. The protocol can be described as the Kripke structure P1 = ⟨AP1, S1, s0, S1, R1, L1, clk1⟩, where AP1 = {Idle1, ROut}, S1 = {s0, s1}, s0 = s0, Σ1 = {clk1, T}, R1 = {s0 T → s0, s0 ROut s1, s1 T → s1, s1 ROut s0}, L1(s0) = {Idle1}, and L1(s1) = {ROut}.

In our setting, all Kripke structures execute synchronously using the same clock. Hence, for the protocols P1 and P2 described above, clk1 = clk2 = clk.

4.2. Composition of Kripke structures

The parallel composition defines the unrestricted composite behavior of two protocols when they are physically connected (without a converter).

Definition 2 (parallel composition). Given Kripke structures P1 = ⟨AP1, S1, s0, S1, R1, L1, clk⟩ and P2 = ⟨AP2, S2, s0, S2, R2, L2, clk⟩, their parallel composition, denoted by P1 ∥ P2, is ⟨AP1∪AP2, S1∪S2, S1∪S2, Σ1∪Σ2, R1∪R2, L1∪L2, clk⟩, where

(i) AP1∪AP2 = AP1 ∪ AP2,

(ii) S1∪S2 ⊆ S1 × S2,

(iii) S0 = (s0, s0),

(iv) Σ1∪Σ2 ⊆ Σ1 × Σ2,

(v) R1∪R2 : S1∪S2 × Σ1∪Σ2 → S1∪S2 is the transition function such that

\[ [s1 \xrightarrow{a} s1'] \land [s2 \xrightarrow{b} s2'] \implies [(s1, s2) \xrightarrow{(a, b)} (s1', s2')], \quad (1) \]

(vi) finally, L1∪L2(s1, s2) = L1(s1) ∪ L2(s2).

Each state s of the parallel composition corresponds to unique individual states s1 and s2 in the protocols, and its labels contain every proposition contained as a label of any of its constituent states. The initial state of the composition is (s01, s02). Each state s = (s1, s2) of the composition makes a transition to a successor state s′ = (s1′, s2′) when both protocol states s1 and s2 make individual transitions to s1’ and s2’, respectively. The transition trigger is obtained by combining the transition triggers of the participating protocols. Note that the set of states S1∪S2 is a subset of the cartesian product of the sets of states S1 and S2. This is because S1∪S2 contains only those states that are reached by both protocols making simultaneous transitions. For the same reason, the event set Σ1∪Σ2 does not contain all elements of the cartesian product of Σ1 and Σ2.

Semantics of the transitions of states in the parallel composition

The parallel composition of two well-formed KS can contain different types of states. The state (s, t) resulting from the parallel composition of two states s and t can be of 4 different types, depending on types of s and t, as shown in Figure 1.

An output state (see Figure 3(a)) in the parallel composition results if each of the states s and t is either an output-only or a delayed-output state. If both s and t are output-only states with (sole) transitions to states s’ and t’ triggered by the outputs π and β, the composite state (s, t) has only one transition to (s’, t’) which results in the emission of the outputs π and β. If s and/or t have tick transitions (in case one or both are delayed-output states), the composite state (s, t) may have additional transitions triggered by a combination of T and the outputs a and b, as shown in Figure 3(a).

An input–output state (see Figure 3(b)) results when one of the states s or t is an input state and the other is an output-only state. Note that for input-output states, all transitions trigger with respect to one KS emitting an output and the other reading an input (or the tick event).

An input-delayed-output state (see Figure 3(c)) is obtained when one of the states s or t is an input state and the other is a delayed-output state. In this case, half of the transitions of the input-delayed-output state corresponds to one KS emitting an output and the other reading an input (or making a T-transition). For example, in Figure 3(c), the transitions (s, t) \xrightarrow{a, T} (s’, t’) and (s, t) \xrightarrow{b, T} (s’, t’′) result in the output τ being emitted. We introduce a function Out : S1∪S2 → 2Σ1∪Σ2, where for any input-delayed-output state s ∈ S1∪S2, Out(s) returns the set of transition triggers that result in the emission of an output (e.g., in Figure 3(c), Out((s, t)) = \{(a, T), (b, τ)\}). The remaining transitions of an input-delayed-output state correspond to one protocol taking a tick transition and the other reading an input (hence delaying the emission of the output). We introduce a function Delay : S1∪S2 → 2Σ1∪Σ2, where for any input-delayed-output state, s ∈ S1∪S2, Delay(s) returns the set of transition triggers that do not result in the emission of an output (e.g., in Figure 3(c), Delay((s, t)) = \{(a, T), (b, T)\}). Note that taking the delay transitions merely delays the emission of an output. For example, if the transition (s, t) \xrightarrow{b, T} (s, t’′)
is taken by the state \((s, t)\) in Figure 3(c), the state \((s, t')\) still refers to the delayed-output state \(s\) which is capable of emitting the output \(\tau\) in the next tick after it is reached.

Finally, an input state (see Figure 3(d)) results when both the states \(s\) and \(t\) are input states. Each transition in the input state is triggered when a pair of inputs (or \(T\)) is read.

Illustration

The parallel composition \(P_1 \parallel P_2\) of \(P_1\) and \(P_2\) in Figure 1 is shown in Figure 4. The initial state \((s_0, t_0)\) corresponds to the initial states \(s_0\) and \(t_0\) of the two protocols and is labelled by \(\text{Idle}_1\) and \(\text{Idle}_2\).

Each transition from the state \((s_0, t_0)\) is triggered when both \(s_0\) and \(t_0\) take their respective transitions. For example, the transition from state \((s_0, t_0)\) to state \((s_1, t_1)\) is triggered when \(s_0\) makes a transition to \(s_1\) (emitting \(\text{enr}\)) and \(t_0\) makes a simultaneous transition to \(t_1\) (reading \(\text{gnt}\)).

All states in the parallel composition presented in Figure 4 are input-delayed-output states as each corresponds to a delayed-output state from \(P_1\) and an input state from \(P_2\) (\(P_1\) has only delayed-output states whilst \(P_2\) has only input states). For the input-delayed-output state \((s_0, t_0)\), \(\text{Out}(s_0, t_0) = \{(\text{req}, T), (\text{req}, \text{req})\}\) while \(\text{Delay}(s_0, t_0) = \{(T, T), (T, \text{req})\}\).

5. SPECIFICATIONS

Specifications of correct interaction between protocols may be formally described using temporal logic. In our approach, \(\text{ACTL} \cup \text{CTL}\), the universal fragment of \(\text{CTL}\) is used. \(\text{ACTL}\) is a branching time temporal logic with universal path quantifiers. As described earlier, \(\text{ACTL}\) is used because it can describe common protocol conversion specifications and at the same time results in a lower worst-case complexity of the conversion algorithm than \(\text{CTL}\).

\(\text{ACTL}\) is defined over a set of propositions using temporal and boolean operators as follows:

\[
\phi \rightarrow P \| -P | f f | f f \phi \land \phi | f f \phi \lor \phi | AX \phi | A(\phi U \rho) | AG \phi.
\]

(2)

Note that, in the above, negation is not applied on temporal and boolean operators. This restriction is due to the fact that the converter generation algorithm uses tableau generation (similar to the local module checking algorithm presented in [3]), where tableau rules can operate only on formulas where negations are applied over propositions.

Semantics of \(\text{ACTL}\) formula, \(\phi\) denoted by \(\llbracket \phi \rrbracket_M\), is given in terms of the set of states in Kripke structure, \(M\), which satisfies the formula (see (3)):

\[
\llbracket p \rrbracket_M = \{s \mid p \in L(s)\}, \quad \llbracket \text{enr} \rrbracket_M = s, \quad \llbracket gnt \rrbracket_M = \emptyset, \quad \llbracket f f \rrbracket_M = \emptyset, \quad \llbracket \text{req} \rrbracket_M = \emptyset, \quad \llbracket \text{req} \rrbracket_M = \emptyset, \quad \llbracket AX \phi \rrbracket_M = \{s \mid \forall s' \rightarrow s' \in \{\phi_M\}\}, \quad \llbracket A(\phi U \psi) \rrbracket_M = \{s \mid \forall t (s = s_1 \rightarrow s_2 \rightarrow \cdot \cdot \cdot)\}, \quad \llbracket AG \phi \rrbracket_M = \{s \mid \forall i (s_i = s_1 \rightarrow s_2 \rightarrow \cdot \cdot \cdot \land \forall i \rightarrow s_i = \phi(i \geq 1)\}.
\]

(3)

A state \(s \in S\) is said to satisfy an \(\text{ACTL}\) formula expression \(\phi\), denoted by \(M, s \models \phi\), if \(s \in \llbracket \phi \rrbracket_M\). We will omit \(M\) from the \(\llbracket \ \rrbracket\) and the \(\models\) relation if the model is evident in the context. The short hand \(M \models \phi\) is used to indicate \(M, s_0 \models \phi\).

The desired behavior of the interaction of the handshake and serial protocols presented in Section 3 is described formally using the following \(\text{ACTL}\) formulas.
In Figure 4, it can be seen that certain paths are inconsistent with the specifications described in Section 3.

For example, the transition from the initial state \((s_0, t_0)\) to \((s_0, t_1)\) results in the serial protocol reading the input req before handshake protocol has emitted it, hence violating the property \(\varphi_1\).

6. PROTOCOL CONVERTERS

The composition \(P_1 \parallel P_2\) (see Figure 4) represents the unconstrained behavior of the protocols including undesirable paths introduced due to mismatches. A converter is needed to bridge the mismatches appropriately. This section formally introduces converters and also the control exerted by a converter over participating protocols.

6.1. I/O relationships between converters and protocols

Firstly, we describe the relationship between the input/output signals of a converter and the participating protocols. Inputs to the converter are outputs from the protocols and vice versa. For example, Figure 1 shows the handshake and serial protocols connected via a converter. The converter reads the outputs \(\text{req}\) and \(\text{gmt}\) of the handshake protocol and emits the signals \(\text{gnt}\) and \(\text{req}\) to be read by the serial protocol. This concept of duality of I/O signals is formalized as follows.

Definition 3 (duality). Given a set of input signals \(\Sigma_i\), a set of output signals \(\Sigma_o\), and two signals \(a\) and \(b\), such that \(\mathcal{D}(a, b) = tt\) if and only if:

(i) \(a \in \Sigma_i\) and \(b \in \Sigma_o\) such that \(b = \overline{a}\), or

(ii) \(a \in \Sigma_o\) and \(b \in \Sigma_i\) such that \(a = \overline{b}\), or

(iii) \(a\) is the tick event \(T\) and \(b = a = T\).

Generalizing to pairs of signals, given two pairs \(A = (a_1, a_2)\) and \(B = (b_1, b_2)\) of signals \(\mathcal{D}(A, B) = tt\) if and only if \(\mathcal{D}(a_1, b_1)\) and \(\mathcal{D}(a_2, b_2)\).

Based on the above definition, each input to the converter is a dual of an output of participating protocols, and vice versa. Also, the tick event \(T\) is its own dual. This is so because a converter does not need to provide any inputs/outputs when protocols make tick-only transitions.

6.2. The role of a converter

A converter \(C\) for the parallel composition \(P_1 \parallel P_2\) of the two protocols is represented as the Kripke structure \((AP_e, SE_e, c_0, S_e, R_e, L_e, clk)\). Many converters that can disable transitions in \(P_1 \parallel P_2\) to ensure the satisfaction of the given ACTL formulas can be envisaged. However, some of these may be incorrect. For example, an output transition in one of the protocols is uncontrollable because a converter cannot prevent it from happening. A converter that attempts to block or disable such a transition is not realizable and hence, deemed incorrect. In our setting, we want to synthesize correct converters. This notion of correctness is formalized by defining a conversion refinement relation between converters and participating protocols. We first provide the intuition.
behind the converter refinement relation by describing the role of a correct converter in controlling a given protocol pair.

Given a converter \( C = (AP_e, SE, \sigma_0, \Sigma_e, R_e, L_e, clk) \), every state \( c \in SE \) controls exactly one state \( s \in S_{1\|2} \) and we say that \( c \) is matched to \( s \). The following restrictions must be satisfied by the matched states \( c \) and \( s \).

1. Every transition out of \( c \) must correspond to a dual transition out of \( s \). A transition \( \sigma' \in C \) is dual to \( \tau' \in S \) if \( D(\sigma, \tau') \) and \( D(\tau', \sigma') \) have dual I/O. Further, the destinations states of the dual transitions must also match. This restriction forms the basis of the control of a converter state over the matched state in the given parallel composition. The converter allows a transition in \( s \) only if \( c \) has a transition that is a dual of the \( s \) transition. In case neither has a dual transition for a given transition in \( s \), the transition is disabled by the converter, nor does the converter disable all transitions of \( s \).

2. Whenever \( s \) is an output state (see Figure 3(a)), the converter is not permitted to disable any transitions of \( s \). In other words, corresponding to each transition of \( s \), \( c \) must have a dual transition. This restriction comes from the fact that each transition of an output state corresponds to the protocols emitting outputs or choosing to remain in the current state. These transitions cannot be disabled by the converter as the protocols do not read any inputs during any transition in an output-state.

3. If \( s \) is an input-delayed-output state (see Figure 3(c)), the converter has to enable precisely one output transition and one delay transition of \( s \). An input-delayed-output state corresponds to a delayed-output state in one protocol and an input state in the other and has two types of transitions. A delayed-output protocol state has two types of transitions: output transitions (returned by the set \( Out \)) and delay transitions (returned by the set \( Delay \)). Output transitions (triggered by the elements of \( Out(s) \)) result in an output being emitted by the protocol while the delay transitions (triggered by the elements of \( Delay(s) \)) result in the protocols delaying the emission of the output to at least the next tick. As the converter cannot force the protocols from emitting an output or taking a delay transition, it must enable one transition each from the sets \( Out(s) \) and \( Delay(s) \).

We now define the conversion refinement relation.

**Definition 4** (conversion refinement relation). Let \( P_1 \parallel P_2 \) be the parallel composition of two protocols \( P_1 \) and \( P_2 \). A state \( s \in S_{1\|2} \) can be represented as the state \( (s_1, s_2) \), where \( s_1 \in S_1 \) and \( s_2 \in S_2 \).

Given a converter \( C = (AP_e, SE, \sigma_0, \Sigma_e, R_e, L_e, clk) \), a relation \( \mathcal{B} \subseteq SE \times S_{1\|2} \) is a conversion refinement relation if for any \((c, s) \in \mathcal{B} \), the following conditions hold.

1. **Enabled transitions.** For all \( c \xrightarrow{\tau} c' \), there exists \( s \xrightarrow{\sigma} s' \), for some \( s, s' \in S_{1\|2} \), such that \( D(\sigma, \tau') \) and \( (c', s') \in \mathcal{B} \).
2. **Output states restriction.** If \( s \) is an output state, then there must exist a \( c \xrightarrow{\tau} c' \) for every \( s \) such that \( D(\sigma, \tau') \) and \( (c', s') \in \mathcal{B} \).
3. **Input-delayed-output states restriction.** If \( s \) is an input-delayed-output state, then \( c \) must have precisely two transitions \( c \xrightarrow{\tau} c' \) and \( c \xrightarrow{\rho} c'' \) that match transitions \( s \xrightarrow{\sigma} s' \) and \( s \xrightarrow{\sigma} s'' \) of \( s \), respectively, such that \( D(\sigma', \sigma'') \), \( D(\sigma', \sigma'') \), \( \sigma' \in Out(s) \), \( \sigma'' \in Delay(s) \), \( (c', s') \in \mathcal{B} \) and \( (c'', s'') \in \mathcal{B} \).

A conversion refinement relation between a converter \( KS \) and a given parallel composition states the constraints on a converter state \( c \) that controls a state \( s \) in the composition.

**Illustration**

Figure 5 presents a converter \( C \) for the composition of handshake-serial protocol pair presented in Figure 4. There exists a conversion refinement relation \( \mathcal{B} \) between the converter and the protocols, where

1. \( \mathcal{B}(c_0, (s_0, t_0)) \): each transition of \( c_0 \) is dual to some transition of \( (s_0, t_0) \) (rule 1) and \( c_0 \) has at least one transition (rule 2). State \( (s_0, t_0) \), being an input-delayed-output state, requires \( c_0 \) to enable one output transition and another tick transition (rule 3). \( c_0 \) satisfies this restriction by having the transition \( (\rho, T) \) \( c_1 \) that reads \( req \) if it is emitted by the protocols and the transition \( (\tau, T) \) \( c_0 \) allowing \( (s_0, t_0) \) to wait;

2. \( \mathcal{B}(c_1, (s_1, t_0)) \): each transition of \( c_1 \) is dual to some transition of \( (s_1, t_0) \) (rule 1) and \( c_1 \) has at least one transition (rule 2). State \( (s_1, t_0) \), being an input-delayed-output state, requires \( c_1 \) to one output transition and another tick transition (rule 3). \( c_1 \) satisfies this restriction by having the transition \( (\sigma, T) \) \( c_2 \) that reads \( gnt \) if it is emitted by the protocols and the transition \( (\tau, T) \) \( c_1 \) allowing \( (s_1, t_0) \) to wait;

3. \( \mathcal{B}(c_2, (s_0, t_1)) \): each transition of \( c_2 \) is dual to some transition of \( (s_0, t_1) \) (rule 1) and \( c_2 \) has at least one transition (rule 2). State \( (s_0, t_1) \), being an input-delayed-output state, requires \( c_2 \) to enable one output transition and another tick transition (rule 4). \( c_2 \) satisfies this restriction by having the transition \( (\rho, T) \) \( c_1 \) that reads \( req \) if it is emitted by the protocols and the transition \( (\tau, T) \) \( c_0 \) allowing \( (s_0, t_1) \) to wait.

We use the conversion refinement relation to define correct converters.
6.3. Definition of converters

Having defined the relationship between the states of a converter and the states of participating protocols, we now define correct converters as follows.

**Definition 5** (converter). A correct converter \( C \) for two protocols \( P_1 \) and \( P_2 \) with parallel composition \( P_1 \parallel P_2 \) is a Kripke structure \( \langle AP_C, S_C, c_0, \Sigma_C, R_C, L_C, clk \rangle \), where

1. \( AP_C = \emptyset \);

2. \( S_C \) is a finite set of states and there exists a conversion refinement relation \( B \) such that for every state \( c \in S_C \), there is a state \( s \in S_{1|2} \) such that \( B(c, s) \);

3. \( c_0 \) is the initial state such that \( B(c_0, s_{1|2}) \);

4. \( \Sigma_C \subseteq \{ a : b \in \Sigma_1 \land D(a, b) \times \{ a : b \in \Sigma_2 \land D(a, b) \} \};

5. \( R_C : S_C \times S_C \to S_C \) is the transition function; and

6. \( L(c) = \emptyset \) for any state \( c \in S_C \).

A converter is a KS whose states are related by a conversion refinement relation to the states of the given parallel composition. Its inputs are the outputs from the parallel compositions and its outputs are the inputs to the parallel composition. Converter states do not have any state labels. The converter also operates on the same clock \( clk \) as the protocols. Note that converters are required to have a transition function that is total with respect to \( S_C \). This means that for every state \( c \in S_C \), there must be at least one transition \( c \xrightarrow{o} c' \) for some \( o \in \Sigma_C \) and \( c' \in S_C \).

**Illustration**

The converter (see Figure 5) for the handshake-serial pair has the following elements.

(i) \( S_C = \{ c_0, c_1, c_2 \} \) with \( c_0 \) as the initial state.

(ii) \( AP_C = \emptyset \), and for any state \( c \) in the converter, \( L(c) = \emptyset \).

(iii) \( \Sigma_C = \{ \{ req, gnt, T \} \times \{ req, gnt, T \} \} \).

(iv) As noted earlier, there exists a conversion refinement relation between the states of \( C \) and \( P_1 \parallel P_2 \).

6.4. Lock-step composition

The control of a converter over a given parallel composition is defined using the // operator as follows.

**Definition 6** (lock-step converter composition). Given the KS \( P_1 \parallel P_2 = (AP_{1|2}, S_{1|2}, s_{0_{1|2}}, \Sigma_{1|2}, R_{1|2}, L_{1|2}, clk) \) and a converter \( C = (AP_C, S_C, c_0, \Sigma_C, R_C, L_C, clk) \), such that there exists a conversion refinement relation \( B \) between the states of the \( C \) and \( P_1 \parallel P_2 \), the lock-step composition \( C // (P_1 \parallel P_2) = (AP_{1|2} \cup S_{1|2}, s_{0_{1|2}}, \Sigma_{1|2} \cup L_{1|2}, \Sigma_{1|2}, R_{1|2}, L_{1|2} // (c, s), clk) \), where

1. \( S_{1|2} = \{ (c, s) : c \in S_C \land s \in S_{1|2} \land B(c, s) \};

2. \( s_{0_{1|2}} = S_{1|2} \) is the initial state, \( s_{0_{1|2}} = (c_0, s_{0_{1|2}}) \);

3. \( R_{1|2} = (\Sigma_C \times S_{1|2} \to S_{1|2}) = (\Sigma_C \times S_{1|2} \to S_{1|2}) \) is the transition function, where for each state \( s_{1|2} = (c, s) \), that has the following transitions:

\[
\begin{align*}
(c, s) & \xrightarrow{o} (c', s') \\
\sigma & \xrightarrow{\sigma} (c', s')
\end{align*}
\]

(4)

(4) \( L_{1|2} // (c, s) = L_{1|2}(s_{1|2}) \).

The lock-step composition ensures that states in the protocols take only those transitions that are allowed by the converter. Each state in the lock-step composition corresponds to a state in the converter and its corresponding state in the parallel composition of the given participating protocols. For example, the initial state of the lock-step...
composition corresponds to the initial state of the converter and the initial state of the given parallel composition. For any state in the lock-step composition, a transition is allowed when its constituent converter state has a transition which is dual to a transition in its corresponding state in the given parallel composition. In other words, when a transition in the converter provides any outputs needed by the participating protocols to trigger a transition in their composition, and the outputs emitted by the protocols are read as inputs by the converter during the same transition, a transition in the lock-step composition triggers. Similarly, if the protocols emit any outputs in a transition or do a tick transition, they are read by the converter in a dual transition. The presence of dual transitions is guaranteed due to the presence of a conversion refinement relation between the states of the converter and the protocols.

Illustration

Figure 6 presents the lock-step composition $\mathcal{C}///(P_1 \parallel P_2)$ for handshake-serial protocol pair presented in Figure 4 and the converter $\mathcal{C}$ presented in Figure 5. The key features of the composition are as follows.

(i) $S_{\mathcal{C}///(1)} = \{ (c, (s_0, t_0)), (c_1, (s_1, t_0)), (c_2, (s_0, t_1)) \}$ with $(c_0, (s_0, t_0))$ as the initial state.

(ii) For any state $(c, s) \in S_{\mathcal{C}///(1)}$, $c \in B$.

(iii) For any state $(c, s) \in S_{\mathcal{C}///(1)}$, $L_{\mathcal{C}///(1)}((c, s)) = L_{1\parallel 2}(s)$.

(iv) Each transition of any state $(c, s)$ in the lock-step is a result of individual dual transitions of $c$ and $s$. For example, the transition $(c_0, (s_0, t_0)) \xrightarrow{\text{req}} (c_1, (s_1, t_0))$ is possible only because the transition $c_0 \xrightarrow{\text{req}} c_1$ is dual to the transition $(s_0, t_0) \xrightarrow{\text{req}} (s_1, t_0)$.

It is important to note that the lock-step composition operator $//$ is different from parallel composition operator $\parallel$ (Definition 2). $//$ provides state-based control to a converter over participating protocols whereas $\parallel$ describes all possible behaviors of the interaction between two given protocols.

The converter presented in Figure 5 can drive the handshake-serial protocol pair to satisfy the system-level properties given in Section 5, or $\mathcal{C}///(P_1 \parallel P_2) \models \phi_1 \land \cdots \land \phi_3$. The next section details the automatic algorithm that is used to generate the above converter for the handshake-serial protocol pair.

The resulting system

The lock-step composition of a given converter and a composition of protocols is a closed system. The protocols read all inputs from the converter whereas the converter reads all its inputs from the protocols. Once composed with a converter, the system does not interact with the external environment.

7. CONVERTER GENERATION USING TABLEAU-BASED MODEL CHECKING

7.1. Overview

The proposed algorithm attempts to automatically generate a converter from a given pair of protocols and a set $\Psi$ of ACTL properties describing the system-level behavior of their interaction. Given protocols $P_1$ and $P_2$, and a set of ACTL properties $\Psi_0$, the converter generation problem is formalized as follows.

$$\exists \mathcal{C} : \forall \varphi \in \Psi_0 : \mathcal{C}///(P_1 \parallel P_2) \models \varphi.$$  (5)

In other words, is there a converter $\mathcal{C}$ for the given protocols $P_1$ and $P_2$ such that in the presence of $\mathcal{C}$, the protocols satisfy all the properties contained in $\Psi_0$?

The proposed approach is based on the local module checking algorithm presented in [3] with nontrivial extensions for use in the convertibility verification domain. Convertibility verification using ACTL specifications is carried out using tableau construction. The conversion methodology can be summarized as follows.

1. Identify the protocols of given IPs and extract their KS representation.
2. Describe system-level properties using the temporal logic ACTL.
3. Employ tableau construction to generate (if possible) a successful tableau given the inputs identified in steps 1 and 2.
4. If no successful tableau can be generated, return to steps 1 or 2 (or both) to modify inputs (weaken ACTL properties or use modified protocols), then repeat step 3.
5. If a successful tableau is generated in step 3, a converter is extracted automatically.

The convertibility verification algorithm is broken into two major parts: tableau construction and converter extraction. Although both these steps are carried out simultaneously, they are provided separately to aid readability.
7.2. **Tableau construction**

7.2.1. **Inputs**

The tableau construction algorithm takes the following two inputs into consideration: $P_1 || P_2$—the composition of participating protocols, and $\Psi_0$—the set of ACTL properties to be satisfied by the interacting protocols.

7.2.2. **Data structure and initialization**

The proposed algorithm is based on tableau construction where a proof structure, called a tableau, is constructed. A tableau is essentially a table of assertions. It is structured in a top-down manner such that an assertion, called a goal, that appears directly above another assertion, called a subgoal, is true only when the subgoals are true. The initial goal (the top-most assertion) for our algorithm is $c_0/s_0 \models \Psi_0$ which requires the existence of a converter state $c_0$ that can guide the initial state $s_0$ of the protocols to satisfy $\Psi$. This goal is then successively resolved into subgoals using tableau rules (to be described later).

In our setting, like in [18], a tableau is represented as an acyclic directed graph where goals are represented as nodes and the edges represent the top-down hierarchy between a goal and its subgoals. A tableau is defined as follows.

**Definition 7** (tableau). Given $P_1 || P_2$, and a set of ACTL properties $\Psi_0$, a tableau $\text{Tab}$ is a labelled acyclic graph $(N, n_0, L)$, where

(i) $N$ is a finite set of nodes of the tableau. Each node $n \in N$ corresponds to a unique state $s \in S_{12}$ and a unique state $c$ in the converter (to be generated) and a set of formula $\Psi$ where each formula $\varphi \in \Psi$ is a subformula of some formula in $\Psi_0$;

(ii) $n_0$ is the root node of the tableau, or the initial goal, which corresponds to the initial state $s_0$ of $P_1 || P_2$, the initial state $c_0$ of the converter to be generated, and the set of formulas $\Psi_0$;

(iii) $L \subseteq N \times N$ is the set of all links (edges) of the tableau.

Each node, that corresponds to states $c$ and $s$ of the converters and protocols, respectively, and a set of formulas $\Psi$, represents the assertion $c//s \models \Psi$. For example, the root node $n_0$, that corresponds to the states $c_0$ and $s_0$ (initial states of the converter and the protocols, resp.) and the set of formulas $\Psi_0$, represents the top-assertion $c_0/s_0 \models \Psi_0$.

A node $n$ may have one or more children (nodes to which it has outgoing edges). Children nodes represent the subassertions that must be met for the assertion represented by $n$ is satisfied. Nodes with one or more children are called *internal nodes*. Nodes with no children are called *leaf nodes*. A leaf node could be a TRUE_NODE (represented by •), or a FALSE_NODE.

A node is *successful* when the assertion it represents is found to be true. A TRUE_NODE is implicitly successful as it represents the assertion $c//s \models \varnothing$. A FALSE_NODE is implicitly unsuccessful because it represents the assertion $c//s \models f f$.

An internal node is successful when all its children nodes are successful. Finally, a tableau is successful when its root node is successful.

The aim of the proposed algorithm is to generate a successful tableau for the inputs $P_1 || P_2$ and $\Psi_0$. During the initialization phase, only the root node $n_0$ of the tableau is created.

7.2.3. **Tableau construction**

After the tableau is initialized, the root node is processed using the tableau rules for converter generation presented in the following equation:

\[
\begin{align*}
\text{emp} & \quad c//s \models \{ \} \\
\text{prop} & \quad c//s \models \{ p \} \cup \Psi, \quad p \in L(s) \\
\text{and} & \quad c//s \models \{ \varphi_1 \land \varphi_2 \} \cup \Psi \\
\text{or} & \quad c//s \models \{ \varphi_1 \lor \varphi_2 \} \cup \Psi \\
\text{unr_{\varphi}} & \quad c//s \models \{ (\varphi \land AXA \varphi) \cup \Psi \} \\
\text{unr_{\exists}} & \quad c//s \models \{ (\exists \varphi \cdot (s, (\varphi \land A \varphi) \cup \Psi)) \} \\
\text{c//s} & \models \Psi
\end{align*}
\]

The tableau rules are of the following form:

\[
c//s \models \Psi \\
\text{c//s} \models \Psi \\
c//s \models \Psi \\
\text{c//s} \models \Psi
\]

In the above, $c, s, \text{and } \Psi$ are the constituent elements of the given node $n$, where $\Psi$ is the set of formulas to be satisfied by $s$ when it is guided (in lock-step fashion) by the converter $c$. $s$ is a state in $P_1 || P_2$ and $s_1, s_2, \ldots, s_n$ are some successor states of $s$, while $c_1, c_2, \ldots, c_n$ are the states of the converter to be generated. Similarly, $\Psi$ is the set of formulas to be satisfied by $s$ whereas $\Psi_1, \Psi_2, \ldots, \Psi_n$ are some derivatives of $\Psi$. The numerator represents the proof obligation (goal) that $s$ in the presence of $c$ must satisfy $\Psi$. To realize the proof, the denominator obligations (subgoals) must be satisfied.

The construction proceeds by matching the current tableau node (initially the root node) with the numerator of a tableau rule and obtaining the denominator which constitutes the next set of tableau nodes. Whenever a tableau rule is applied to a node (goal) to create new nodes (subgoals), the node has an edge to each such newly created node. Equation (6) presents the tableau rules for convertibility verification using ACTL specifications.

The rule emp corresponds to the case when there is no obligation to be satisfied by the current state of the protocols; any converter is possible in this case, that is, the converter allows all possible behavior of the parallel composition at
state $s$. As this scenario describes a TRUE_NODE (●), which is implicitly successful, no further processing is possible.

The pr-op rule states that a converter is synthesizeable only when the proposition is contained in the labels of the parallel composition state $s$; otherwise there exists no converter. Once the propositional obligation is met, the subsequent obligation is to satisfy the rest of the formulas in the set $Ψ$. For example, if $Ψ$ contains the formulas $p$, $q$, and $r$, and if the $p$ is a proposition, the rule checks if the parallel composition state $s$ is labelled by $p$. If this is indeed the case, the denominator then requires that the formulas $q$, and $r$ are satisfied by the state.

The $∧$-rule states that the satisfaction of the conjunctive formula depends on the satisfaction of each of the conjuncts. The $∨$-rules are the duals of $∧$-rule. The rule $unr_{au}$ depends on the semantics of the temporal operator $AU$. A state is said to satisfy $A(φ U ψ)$ if and only if it either satisfies $ψ$ or satisfies $φ$ and evolves to new states each of which satisfies $A(φ U ψ)$. Similarly, $AGφ$ is satisfied by states which satisfy $φ$ and whose all next states satisfy $AGφ$ (Rule $unr_{ag}$).

Finally, $unr_{x}$ is applied when the formula set in the numerator $Ψ$ consists formulas of the form $AXφ$ only. Satisfaction of these formulas demands that all successor states of the $c//s$ must satisfy every $φ$ where $AXφ ∈ Ψ$, that is, $c//s$ satisfies all elements of $Ψ_{AX}$. A converter controls the parallel composition through implicit disabling induced by this rule. The unrestricted behavior of the protocols (where $c$ allows all the transitions from $s$) may not be able to satisfy this obligation as one or more successors may fail to satisfy the commitments passed to them. However, we can check if a subset of the successors satisfies these commitments. If a subset of successors satisfies these future commitments, the converter can disable all other transitions of $s$ that lead to states that are not contained in this subset.

In order to identify the subset of successors that satisfies all future commitments of the current state, we pass these commitments to each successor of the current state. For $k$ possible successors, we need to perform $k$ passes of this step. During each step, a new successor is chosen and the future commitments are passed to it. If it returns success, it is added to the subset. Once all successors have been checked, we return success if the subset is nonempty, and if it satisfies the well-formedness conditions described earlier in Section 6.2. In case the subset is empty, we note that there is no successor of the state $s$ that can fulfill its future commitments, and we return failure (an unsuccessful tableau).

7.2.4. Termination: finitizing the tableau

It is important to note that the resulting tableau can be of infinite depth as each recursive formula expression $AU$ or $AG$ can be unfolded infinitely many times.

This problem due to the unbounded unfolding of the formula expressions can be addressed using the fixed-point semantics of the formulas $AGφ$ and $A(φ U ψ)$. The former is a greatest fixed-point formula while the latter is a least fixed-point formula,

$$AGφ ≡ Z_{AG} = ν ϕ ∧ AXZ_{AG},$$

$$A(φ U ψ) ≡ Z_{AU} = µ ψ ∨ (φ ∧ AXZ_{AU}).$$  \hspace{1cm} (8)

The greatest (least) solution for $Z_{AG}$ ($Z_{AU}$) is the semantics of $AG(φ)$. It can be shown (details are omitted) that satisfaction of the greatest fixed-point formula is realized via loops in the model. Least fixed-point formulas require that the paths satisfying the formulas must have finite length. For these formulas, if a tableau node is revisited, then it can be inferred that the LFP formula is not satisfied. As such, if a tableau node $c′//s ⇒ Ψ$ is visited and there exists a prior node $c//s ⊨ Ψ$ (the same tuple $s$ paired with the same $Ψ$ is seen in a tableau path), we check whether there exists a least fixed-point formula $AU$ in $Ψ$. If such a formula is present, the tableau path is said to have resulted in an unsuccessful path (FALSE_NODE is returned). Otherwise, the tableau path is successfully terminated by equating $c′$ with $c$ (a loop in the converter is generated), and TRUE_NODE is returned.

We now look at how the proposed tableau construction algorithm is implemented.

7.3. Converter generation algorithm

Algorithm 1 shows the recursive converter generation procedure. Given a state $s$ of the parallel composition, a set of subformulas $FS$, and a history of all previously visited tableau nodes $H$ (all nodes visited on the path from the root node to the current node) used for finitizing the tableau (as discussed in Section 7.2), the algorithm first checks if there are no commitments to be met. In that case, it returns success (●), otherwise it creates a new node with respect to the state $s$ and the set of formulas FS. It then checks if a node with the same elements (state and formulas) has been visited before. If such a node is found, and FS contains an $AU$ formula, the algorithm returns failure (FALSE_NODE). Otherwise it returns success (see notes on finitizing the tableau in Section 7.2) that results in a loop in the converter. If no matching node is found, the current node is added to the set of visited nodes. We remove a formula $F$ from $FS$. Depending on the type of the formula, the corresponding tableau rule is applied by calling Algorithm 1 recursively. If the recursive call returns a non-FALSE_NODE, we return success (by adding the node returned by the recursive call as a child of the current node). Consider for example the handling of a disjunction $φ ∨ ψ$. The algorithm first checks if the state satisfies $φ$ along with any other commitments (subformulas) left after $F$ was removed. If a successful tableau can be generated, the node returns success. Otherwise, we check if the state satisfies $ψ$ along with the remaining subformulas and returns success if a successful tableau can be generated. If however neither tests returns success, the node returns failure.

If FS contains only future commitment (AX formulas), the algorithm proceeds as follows. Firstly, the node is marked as an $X_NODE$. An $X_NODE$ refers to an internal node in the tableau that has been extended using the $unr_{x}$ rule (see (6)). The future (AX) commitments are passed to each successor of $s$. If success is returned, we create a link from the current node to this newly created node (corresponding to the selected successor). If failure is returned, we check if the disabling of a transition to this successor would result in the breach of a rule of the converter refinement relation.
(1) if \( FS = \emptyset \) then
(2) \( \text{return TRUE\_NODE} \)
(3) end if
(4) curr = createNODE(s, FS);
(5) if \( \text{anc} \in H = \text{curr} \) then
(6) if \( \text{FS} \) contains \( AX \) formulas then
(7) \( \text{return FALSE\_NODE} \)
(8) else
(9) Remember link between \( \text{curr} \) and \( \text{anc} \)
(10) curr.add(\text{CHILD}(\text{TRUE\_NODE}))
(11) \( \text{return curr} \)
(12) end if
(13) end if
(14) \( H_1 = H \cup \{ \text{curr} \} \);
(15) if \( \text{FS} \) contains a formula \( F \) which is not of type \( AX \) then
(16) \( \text{FS}_1 := \text{FS} \setminus F \); \( \text{Node} \) \( \text{ret} := \text{FALSE\_NODE} \)
(17) if \( F = \text{TRUE} \) then
(18) \( \text{ret} := \text{isConv} (s, \text{FS}_1, \text{H}_1) \)
(19) else if \( F = p \ (p \in AP) \) then
(20) if \( p \) is satisfied in \( s \) then
(21) \( \text{ret} := \text{isConv} (s, \text{FS}_1, \text{H}_1) \)
(22) end if
(23) else if \( F = \neg p \ (p \in AP) \) then
(24) if \( p \) is not satisfied in \( s \) then
(25) \( \text{ret} := \text{isConv} (s, \text{FS}_1, \text{H}_1) \)
(26) end if
(27) else if \( F = \varphi \land \psi \) then
(28) \( \text{ret} := \text{isConv} (s, \text{FS}_1 \cup \{ \varphi, \psi \}, \text{H}_1) \)
(29) else if \( F = \varphi \lor \psi \) then
(30) \( \text{ret} := \text{isConv} (s, \text{FS}_1 \cup \{ \varphi \}, \text{H}_1) \)
(31) if \( \text{ret} = \text{FALSE\_NODE} \) then
(32) \( \text{ret} := \text{isConv} (s, \text{FS}_1 \cup \{ \psi \}, \text{H}_1) \)
(33) end if
(34) else if \( F = \text{AG} \varphi \) then
(35) \( \text{ret} := \text{isConv} (s, \text{FS}_1 \cup \{ \varphi \ \text{AXA} \varphi \}, \text{H}_1) \)
(36) else if \( F = A(\varphi \cup \psi) \) then
(37) \( \text{ret} := \text{isConv} (s, \text{FS}_1 \cup \{ \psi \ \text{AXA} (\varphi \cup \psi) \}, \text{H}_1) \)
(38) end if
(39) if \( \text{ret}! = \text{FALSE\_NODE} \) then
(40) curr.add(\text{CHILD}(\text{ret}))
(41) end if
(42) \( \text{return ret} \)
(43) end if
(44) curr.type := \text{X\_NODE}
(45) \( \text{FS}_{AX} = \{ \varphi \mid \text{AX} \varphi \in \text{FS} \} \)
(46) for each successor \( s' \) of \( s \) do
(47) if \( \text{N} := \text{isConv} (s', \text{FS}_{AX}, \text{H}_1) \neq \text{FALSE\_NODE} \) then
(48) curr.add(\text{CHILD}(\text{N}))
(49) else if \( \text{Transition to} \ s' \ \text{can not be disabled (As disabling it would violate the conversion refinement rules)} \) then
(50) \( \text{return FALSE\_NODE} \)
(51) end if
(52) end for
(53) return curr

\textbf{Algorithm 1: NODE isConv}(s, FS, E).

(1) Create new map \( \text{MAP} \)
(2) Create new map \( \text{PURE\_MAP} \)
(3) initialState = \text{extractState}(t, \text{rootnode});
(4) \( \text{return initialState} \)

\textbf{Algorithm 2: STATE extractConverter(Tableau t).}

(1) if NODE is present in MAP then
(2) \( \text{return map.get(NODE)} \)
(3) else if NODE is an internal node then
(4) \( \text{MAP.put(NODE, extract(NODE.child))} \)
(5) \( \text{return MAP.get(NODE)} \)
(6) else if NODE is TRUE\_NODE then
(7) if NODE is related to an ancestor \( \text{anc} \) then
(8) \{see line 9 of isConv\}
(9) \( \text{MAP.put(NODE, getPureState(\text{ANC} \text{.a})} \)
(10) \( \text{return MAP.get(NODE)} \)
(11) else
(12) \( \text{MAP.put(NODE, getPureState(NODE.a)} \)
(13) \( \text{return MAP.get(NODE)} \)
(14) end if
(15) else if NODE is of type \text{X\_NODE} then
(16) create new converter state \( c \)
(17) \( \text{MAP.put(NODE, c)} \)
(18) for each linked NODE' of NODE do
(19) State \( c' = \text{extract(NODE')} \)
(20) add transition \( c' \rightarrow \) \( c \) where \( \text{NODE} \rightarrow \text{NODE'\_a} \) and \( \text{D}(\sigma, \sigma') \).
(21) end for
(22) \( \text{return MAP.get(NODE)} \)
(23) end if

\textbf{Algorithm 3: NODE extract(NODE).}

(see Definition 4). For example, if the current successor is reached via an output transition, the transition to it cannot be disabled. In such cases when a successor does not meet future commitments and we cannot disable transitions to them, we return failure. The algorithm returns success when one or more successors of a satisfy the future commitments (and the converter refinement relation holds if a converter state enables these transitions in s).

7.4. Converter extraction

If a successful tableau is constructed, the converter is extracted by traversing the nodes of the tableau as shown in extractConverter algorithm (Algorithm 2). Firstly, it creates a map \( \text{MAP} \). \( \text{MAP} \) is essentially a lookup table that relates nodes in the tableau (keys) to states in the converter being generated (values). This map is initially empty. We then pass the tableau’s root node to the recursive procedure extract (Algorithm 3) which processes a given node as follows.

(1) If NODE is already present in MAP as a key, return the converter state associated with it.
(1) if $s$ is present in PURE_MAP then
(2) return PURE_MAP.get($s$)
(3) else
(4) create new converter state $c$
(5) PURE_MAP.put($s$,$c$)
(6) for each successor $s'$ of $s$ do
(7) State $c' =$ getPureState($s'$)
(8) add transition $c \xrightarrow{\sigma} c'$ where $\sigma$ is the label of the transition from $s$ to $s$
(9) end for return $c$
(10) end if

Algorithm 4: NODE getPureState($s$).

(2) If NODE is an internal node, the converter state corresponding to this node is the converter state extracted with respect to its child. An internal node always has one child as it is expanded by the application of a tableau rule other than unReach.

(3) If NODE is of type TRUE_NODE, and it is related to an ancestor node (see line 9 of isConv), the converter state corresponding to the node is the same as the converter state corresponding to its linked ancestor.

(4) If NODE is of type TRUE_NODE but is not related to any ancestors, the converter allows all transitions in $P_1 \cup P_2$ from this state onwards. We return a converter state that allows all transitions in the state corresponding to NODE and any of its successors (see Algorithm 4).

(5) If NODE is of type X_NODE, we create a new converter state corresponding to the node. The created state $c$ contains transitions to each state corresponding to each linked child of the X_NODE.

Illustration

This section presents the steps involved in the tableau construction and converter extraction for the handshake-serial example in Figure 1.

The tableau construction for the handshake-serial example starts at the construction of the root node of the tableau corresponding to the initial state $(s_0,t_0)$ of $P_1 \cup P_2$ (see Figure 2), and the set $FS$ (see Section 5) of system-level properties to be satisfied. The root node is created when these arguments are passed to the recursive algorithm isConv.

The processing of the root node is shown in Table 1. The algorithm proceeds as follows. Given the root node (node 0), the algorithm removes one formula $F = \text{AG}(Idl e_1 \land Idl e_2 \Rightarrow \text{AX}(\text{ROut} \lor \neg \text{RIn}))$ from the set NODE.FS. Then, $F$ is broken down into simpler commitments $(Idl e_1 \land Idl e_2 \Rightarrow \text{AX}(\text{ROut} \lor \neg \text{RIn})) \land \text{AXAG}(Idl e_1 \land Idl e_2 \Rightarrow \text{AX}(\text{ROut} \lor \neg \text{RIn}))$. These simpler commitments are then reinserted into the set $FS$ of a new node (node 1 as shown in Table 1) along with all remaining commitments in node 1.FS, and a recursive call to isConv is made. The newly created node is added as a child node of NODE if it returns success. Whenever $F$ is a propositional formula, it can be checked against the labels of the state $(s_0,t_0)$. The process of removing one formula and reinserting its subformulas stops when NODE contains no formulas or only $AX$-type formulas (node 9).

When only $AX$ formulas are left (node 9 in Table 1), the node is labelled as an X_NODE (line 44). At this stage, for every successor of $(s_0,t_0)$, the algorithm makes a recursive call to itself. During each call, the arguments to isConv are a unique successor of $(s_0,t_0)$, and all the commitments to be satisfied by the successor (if it is enabled by the converter). For calls that return success, their corresponding nodes are added as children of node 9.

The above process continues until the recursive call made by the root node returns. For the given example, this call returns success, which means that a successful tableau has been constructed. Using the converter extraction algorithm described earlier, the tableau yields the converter presented in Figure 5.

7.5. Complexity

The tableau considers all possible subformulas of the given set of desired properties. In the worst case, each subformula can be paired with every possible state in the protocol pair. The complexity of the tableau construction is therefore $O(|S| \times 2^{|\phi|})$, where $S$ is the number of states in the protocol pairs and $|\phi|$ is the size of the formula(s) used for conversion.

The complexity differs from model checking [18, 19]. The complexity for both CTL and ACTL model checking algorithms is $O(|S| \times |\phi|)$, where $\phi$ is the size of the given formula. The reason for this difference arises from the handling of conjunctions of disjunctive formulas in model checking and our algorithm. In model checking, if a state is expected to satisfy a formula $(a \lor b) \land (c \lor d) \land (e \lor f)$, the algorithm first computes the states that satisfy the subformulas $a, b, c, d, e, f, (a \lor b), (c \lor d), (e \lor f)$ of the given formula before computing the states that satisfy the given formula. Hence, each subformula is only checked once. However, in our approach, the increase in complexity occurs due to the fact that all possible subformulas (of the given set of formulas) are considered for every node.

Table 1: Nodes with the attributes $s = (s_0,t_0), c = c_0$.

<table>
<thead>
<tr>
<th>Node</th>
<th>$\Psi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${\text{AX}((\text{RIn} \land \neg \text{ROut}) \Rightarrow \text{AX}(\neg \text{Idle}_1 \lor \neg \text{Idle}_2)), \text{AG}(\text{RIn} \land \text{ROut} \Rightarrow \text{AX}(\neg \text{Idle}_1 \lor \neg \text{Idle}_2)), \text{AG}(\text{Idle}_1 \land \text{RIn} \Rightarrow \text{AX}(\text{RIn} \lor \neg \text{ROut}))}$</td>
</tr>
<tr>
<td>1</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
<tr>
<td>2</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
<tr>
<td>3</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
<tr>
<td>4</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
<tr>
<td>5</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
<tr>
<td>6</td>
<td>${\text{AXAG}<em>{\text{RIn}}, \text{AXAG}</em>{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}<em>2}, \text{AXAG}</em>{\text{RIn}}, \text{AXAG}_{\text{Idle}<em>1}, \text{AXAG}</em>{\text{Idle}_2}}$</td>
</tr>
</tbody>
</table>
For example, Figure 7 shows a node which requires the conjunction of three disjunctive formulas to hold on a state in the protocols. As there is no particular order in which formulas are picked from the node's commitment set FS (containing the three formulas), we consider the order given in Figure 7. Formulas $a, \ldots, f$ could be any type of ACTL formulas. Each node is numbered in the order visited. Initially, the disjunction $a \lor b$ is removed from $\psi$ and a child node checking only $a$ and the other formulas is created (node 2). In node 2, the formula $c \lor d$ is chosen which results in the creation of node 3 which checks $a$, $c$, and $e \lor f$. Similarly, node 3 leads to node 4 which checks $a$, $c$, and $e$. At this stage, as all remaining formulas are propositional, no further breaking up can be carried out. Assuming that $a$, $c$, and $e$ are propositions, it is possible for the algorithm to check their satisfaction linearly. In case any one is not satisfied, we return failure which results in the creation of node 5 which checks $a$, $c$, and $f$. Given the order in Figure 7 and assuming that none of the propositional formulas are satisfied by the corresponding nodes, the algorithm terminates after it has checked all possible combination of disjuncts. It can be seen that the number of nodes created is exponential to the number of disjunctive formulas.

### 7.6. Soundness and completeness

The following theorem follows from the above discussion. The proof of the theorem is provided in the appendix.

**Theorem 1** (sound and complete). There exists a converter $C$ to control two given KS $P_1$ and $P_2$, such that the resulting system $C/(P_1 \parallel P_2)$ satisfies a set FS of ACTL formulas, if and only if isConv ($s_{012}$, FS, $\emptyset$) does not return FALSE_NODE.

### 8. RESULTS

A convertibility verification tool using tableau construction has been implemented using Java. The implementation takes as input the Kripke structure representation of two protocols $P_1$ and $P_2$ and a set $\Psi$ of ACTL properties from the user. The Kripke structure representation of a protocol is extracted automatically from a given NuSMV description of an IP. For this purpose, a modified version of the NuSMV model checking tool is employed [20]. Given these inputs, the algorithm constructs a tableau after computing the parallel composition $P_1 \parallel P_2$. If a successful tableau is created, a converter, represented as a Kripke structure, is automatically generated. This converter is guaranteed to bridge mismatches between the two protocols.

Table 2 shows the results obtained from the protocol conversion of various examples. Problems 1–6 are well-known protocol mismatch problems explored by earlier works such as [4, 12]. Problems 7–9 are derived from well-known NuSMV examples [20]. These examples were created by introducing commonly-encountered control mismatches, such as incorrect signal exchange sequences and lack of mutual exclusion, into the systems. Problems 10–14 are SoC examples that were chosen to show the applicability of our approach for SoC design. Each of these SoC problems modeled control mismatches between the AMBA bus and a peripheral. Different version of AMBA, namely the Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advances Peripheral Bus (APB) were used. Problems 12–15 involve conversion between more than 2 IPs that is achieved by extending the proposed framework. The proposed extension to handle “n”-protocols is presented in [21]. This extension involves the formulation of composition rules for multiple Kripke structures, and new constraints for converters (conversion refinement relation) to extend their control to multiple protocols. The conversion algorithm is then extended such that its input KSs $P_1$ and $P_2$ are parallel compositions of multiple protocols.

The first three columns of Table 2 contain the ID, description, and size (number of states) of the participating protocols. The intuitive description of the ACTL properties used for each problem was shown in the fourth column. For most of the problems presented in Table 2, our algorithm was...
able to generate a converter to control the participation to satisfy the given ACTL properties.

There was one case (Problem 11) when the algorithm failed to generate a converter. It failed because of the inability of the AMBA APB to allow burst transfers (multiple operations per activation of a master). In cases where the algorithm fails (an unsuccessful tableau is returned), it is required that the inputs (IPs and/or specifications) be modified manually in order to bridge mismatches between these protocols. To carry out this manual modification, the unsuccessful tableau returned by the algorithm can be used (as a counterexample) to find the exact reason (state, subformula) for failure.

The significance of these results is explained as follows. For problems 1–6, the use of ACTL specifications resulted in converters that were similar in size than those generated when automata-based specifications were used (in [4, 12]). This shows that ACTL is powerful enough to describe most commonly-encountered specifications used for conversion. ACTL also has the additional benefit of being succinct and more intuitive to write than automata-based properties for many problems. For example, to guide a 14-process system to have mutual exclusion (Problem 15), the resulting automaton that describes such interaction will be very complex and difficult to write while one can write simple ACTL properties that describe such interaction. In addition, a user may provide multiple properties and IPs during conversion using the proposed algorithm, a feature that is not offered by earlier works [4, 6]. Furthermore, our approach is capable of handling bidirectional communication between IPs, which cannot be handled using approaches such as [4, 12]. Finally, specifications in our setting are described using temporal logic properties whereas other approaches like [4, 12] use automata-based specifications. The use of temporal logic allows us to write complex specifications succinctly by breaking them down into multiple properties. Furthermore, in addition to mismatch resolution, the conversion algorithm can be used to enforce additional properties in the converted system.

### 9. CONCLUSIONS

Protocol conversion to resolve mismatches between IPs is an active research area. A number of solutions have been proposed. Some approaches require significant user input and guidance, while some only partly address the protocol conversion problem. Most formal approaches work on protocols that have unidirectional communication and use finite state machines to describe specifications.

In this paper, we propose the first temporal logic-based approach to protocol conversion that can automatically generate a converter given the protocols of mismatched IPs and specifications described as ACTL formulas. The proposed algorithm uses tableau generation and attempts to construct a tableau given the above inputs. If the algorithm succeeds (a successful tableau is constructed), a converter is automatically generated. This converter is guaranteed to control the given IPs such that the required ACTL properties are satisfied.

The approach is shown to be sound and complete and is capable of handling many common mismatches that existing conversion techniques can address. In addition, ACTL formulas are easier to write for complex specifications.
(1) process FS till no conjunctions, $\mathsf{A}G$ and $\mathsf{AU}$ formulas remain.
(2) if FS contains no disjunctions then
(3) return FS
(4) end if
(5) Pick a disjunction $F = \phi \vee \varphi$ from FS
(6) return extractSets(FS\(\cdot F + \phi\)) \(\cup\) extractSets(FS\(\cdot F + \varphi\))

**Algorithm 5:** FormulaSets extractSets (FS).

as compared to automata-based specifications. The proposed technique can handle bidirectional communication between protocols and can also ensure that the converted system satisfies additional temporal logic constraints. All these features are not provided by any existing approaches to protocol conversion.

The future direction for this work involves the extension of the approach to handle data and clock mismatches between IPs and to provide a correct-by-construction design of the approach to handle data and clock mismatches.

**APPENDIX**

In order to prove Theorem 1, we first prove the following lemmas.

**Lemma 1** (termination). A call to $\isConv (s', H', E')$ always terminates.

**Proof.** The proof of this lemma follows from an analysis of recursion in $\isConv$. The following observations can be made.

(i) Whenever $\isConv$ is called, if a node with the same attributes ($s'$ and $\text{FS}'$) exists in the history set $H$, then the call returns without further recursion. This observation can be verified by checking that the newly created node $\text{curr}$ is only added to the tableau in line 13, after the history set $H$ is checked for the presence of a node with the same attributes as $\text{curr}$. If the history set does contain such a node, we either return $\text{FALSE\_NODE}$ or $\text{curr}$ but do not call $\isConv$ any further.

(ii) Whenever a recursive call to $\isConv$ is made, and no node with the same attributes to $\text{curr}$ is found in the history set, the node created in the current call is always added to the history set $H'$ before a recursive call is made (see line 13).

(iii) The number of states in $P_1 \parallel P_2$ and the maximum number of valuations for the sets $\text{FS}'$ are finite in the following manner.

(a) $P_1 \parallel P_2$ has a finite number of states by definition.

(b) $\text{FS}'$ cannot have a valuation outside the $2^{\mid \text{FS}}\mid$ possible subsets of subformulas of $\text{FS}$.

(iv) As each node corresponds to a combination of the following: a state of $P_1 \parallel P_2$, and a subset $\text{FS}'$ of subformulas of $\text{FS}$, both of which have finite valuations, the number of nodes that can be created in $\isConv$ is finite.

As described above, the number of nodes that can be created by $\isConv$ is finite. Furthermore, $\isConv$ stops recursing if a newly created node is already present in the history set, preventing duplicating nodes along all paths. Hence, it is proved that a call to $\isConv$ always terminates.$\square$

**Lemma 2** (sufficient condition). If $\isConv(s_{012}, \text{FS}, \emptyset)$ returns a non-$\text{FALSE\_NODE}$, there exists a correct converter $\mathcal{C}$ such that $\mathcal{C}/P_1 \parallel P_2 \models \text{FS}$.

**Proof.** In order to prove the above lemma, we assume that we have been given a tableau with root node $n_0$ corresponding to the initial state $s_{012}$, and we generate a converter from the tableau and show that the converter helps satisfy the property.

We first note the following characteristics of a successful tableau return by $\isConv$.

(1) A node can never have a false node as a child. A leafnode results when the algorithm finds the same node in the history set. Hence, no further recursion is done and the leafnode itself has no children. An internal node makes a recursive call to $\isConv$, and if that call returns a false node, the calling node does not add the false node to its children list.

(2) A node is present in the tableau if and only if it returned a nonfalse node: if a call to $\isConv$ returns false node, the node is not added to the tableau. Hence, all nodes present in the tableau reachable from $n_0$ returned a nonfalse node.

(3) All leafnodes and true nodes have no children: these nodes are formed when there are no subformulas to satisfy ($\text{FS} = \emptyset$) or when a node with the same attributes is found in history. In both these cases, no further calls to $\isConv$ are made.

(4) All internal non-$\text{X\_NODE}s$ have one child only.

(5) An $\text{X\_NODE}$ may have more than one child.

(6) All internal nodes lead to an $\text{X\_NODE}$, true node, or a leafnode: given a set of formulas $\text{FS}$, the child of $\text{curr}$, if $\text{curr}$ is an internal node, will have the same commitments as $\text{curr}$ except that one of the formulas is broken into smaller commitments. If this process is repeated infinitely often, we reach a fix-point where no formulas in $\text{curr}$ can be broken any further. At this stage, we can only have $\mathsf{AX}$ formulas in $\text{FS}$ which results in the formation of an $\text{X\_NODE}$. If $\text{FS}$ contains no formulas, we form a true node. It is also possible that while breaking down formulas from a parent node to a child node (where both are internal nodes), we may find that a matching node is already present in the history set $H$. In that case, we finitize
the tableau by returning the current node \texttt{curr} and marking it as an internal node.

(7) All leafnodes lead to an \texttt{X\_NODE} before they are visited again: a leafnode points to a node in history, which is an ancestor of the leafnode. That ancestor node cannot be a leafnode because otherwise there would have been no path from the ancestor to the current leafnodes. Furthermore, starting from the ancestor node, none of its descendant nodes can be leafnodes (except the current one) because otherwise a path from that descendant to the leafnode would not have been possible. The commitments in the leafnode are the same as those of the matching ancestor. Now, the child of the ancestor must contain simpler commitments in its formula set than the ancestor. It can be seen that the commitments of an internal node are never repeated by their descendant nodes because of the successive breaking down of commitments until only AX commitments remain. Hence, the ancestor must eventually lead to an \texttt{X\_NODE} (fix point where all formulas except AX commitments). It cannot come to the leafnode without an \texttt{X\_NODE} in between because that would require that the ancestor and leafnodes have different attributes (formula sets) which is not possible.

Given the above observations, we now extract a converter that satisfies \texttt{FS} starting from the root node using the algorithm \texttt{extractConverter}, \texttt{extract}, and \texttt{getPureState} given earlier.

\texttt{extractConverter} creates two global variables \texttt{MAP} and \texttt{PURE\_MAP} that store all converter states that are generated. It then calls the recursive algorithm \texttt{extract} and passes the root node of the successful tableau obtained by \texttt{isConv}.

In algorithm \texttt{extract}, starting from the root node \texttt{n0}, we recurse until we reach an \texttt{X\_NODE} or a \texttt{●}. We cannot come to a leafnode before reaching an \texttt{X\_NODE} from the root node as per the above observations. Furthermore, we can only reach a single \texttt{X\_NODE} as all internal nodes starting from \texttt{n0} can only have one child each.

Corresponding to the \texttt{X\_NODE}, we create the initial node \texttt{c0} of the converter. To \texttt{c0}, we now add all states obtained by recursively calling \texttt{extractConverter} on each of the children of the \texttt{X\_NODE}.

In case we reach a true node, all commitments in \texttt{FS} have been satisfied and there are no future commitments to be satisfied. Hence, we now use \texttt{getPureState} to allow the converter to enable all transitions in the parallel composition from this point onwards.

Once the above converter is extracted, we can prove that it indeed satisfies all formulas \texttt{FS} as follows. From any node \texttt{curr} in the tableau, we may eventually reach an \texttt{X\_NODE curr\_ax} (possibly through a leafnode) or a true node. In case we reach a true node, there are no future commitments to satisfy. Furthermore, all present state commitments must be satisfied in the current node otherwise we could not have reached a true node (each present state commitment is removed from \texttt{FS} as it is satisfied by the current node).

If we instead reach an \texttt{X\_NODE}, it can only contain AX commitments. We can see that all the present state commitments in \texttt{FS} of \texttt{curr} are satisfied by the state \texttt{curr\_state}.

Now, we create a converter state \texttt{c} for each \texttt{X\_NODE}. Furthermore, \texttt{c} contains one node corresponding to each child of the \texttt{X\_NODE}. An \texttt{X\_NODE} contains a child only if all AX commitments present in the present node are satisfied by the successor corresponding to the child. Each child can only be present in the tableau because it returned success. For each of this children, we create a further successor of \texttt{c} and repeat this process until all nodes have been processed.

We can see that each state in the converted system satisfies all its present commitments and its enabled successors satisfy all their future commitments.

\begin{lemma}[correct converters] Any converter \texttt{C} obtained from the tableau returned by the call \texttt{isConv} \((s012,FS,φ)\) is a correct converter.
\end{lemma}

\textbf{Proof.} The proof of the above lemma follows from the proof of the previous lemma which shows that a converter state extracted from an \texttt{X\_NODE} always ensures conversion refinement between itself and the state corresponding to the \texttt{X\_NODE}.

\begin{lemma}[necessary condition] If there exists a converter \texttt{C} such that \texttt{C//P1||P2 |= FS, isConv} \((s012,FS,φ)\) returns a non-\texttt{FALSE\_NODE}.
\end{lemma}

\textbf{Proof.} We assume that we have been given a converter \texttt{C} under which \texttt{P1||P2} satisfies all commitments \texttt{FS}. Furthermore, we assume that \texttt{isConv} \((s012,FS,φ)\) returns a \texttt{FALSE\_NODE}.

We now show that above statements are contradictory.

We first process the set \texttt{FS} successively in the following manner. Each formula \texttt{F} in \texttt{FS} is processed as follows.

\begin{enumerate}
  \item \texttt{F} is \texttt{tt, ff, proposition, negated proposition, disjunction, or AX formula}, it is not processed further.
  \item \texttt{F} = \texttt{ϕ ∨ φ}: after processing, \texttt{FS} = \texttt{FS} - \texttt{F} \cup \{\texttt{ϕ}, \texttt{φ}\}.
  \item \texttt{F} = \texttt{AX}\texttt{ϕ}: after processing, \texttt{FS} = \texttt{FS} \cup \{\texttt{ϕ ∧ AXAϕ}\}. Again, \texttt{ϕ} can be processed further depending on its type whilst \texttt{AXAϕ} is an \texttt{AX} formula.
  \item \texttt{F} = \texttt{A(ϕ U φ)}: after processing, \texttt{FS} = \texttt{FS} \cup \{\texttt{ψ} \lor (\texttt{ϕ ∧ AXAϕ})\}, which is a disjunction.
\end{enumerate}

Once the above processing is completed, \texttt{FS} will contain only propositions (including \texttt{tt} and \texttt{ff}), negated propositions, disjunctions, or AX formulas.

Using the algorithm \texttt{extractSets}, we remove all disjunctions by creating multiple copies of \texttt{FS} (Algorithm 5). Given a formula \texttt{ϕ ∨ φ} in \texttt{FS}, we create two copies of \texttt{FS}, each containing all formulas in \texttt{FS} except the disjunction plus one of the operands of the disjunction. We process \texttt{FS} until all disjunctive formulas are removed, and a set \texttt{SetOfFormulaSets} containing multiple formula sets. Furthermore, we remove from \texttt{SetOfFormulaSets} all sets that contain \texttt{ff}, because such a set cannot be satisfied by any state.
in a Kripke structure. Furthermore, we remove from all sets in \(\text{SetOfFormulaSets}\) the formula \(tt\) because it is implicitly satisfied by all states in any given Kripke structure.

Each set in \(\text{SetOfFormulaSets}\) contains only propositions, negated propositions, and disjunctions. For a state in a Kripke structure to satisfy the original formula set \(FS\), it must satisfy at least one of the sets contained in \(\text{SetOfFormulaSets}\). The state must satisfy all propositions and negated propositions, and its successors must satisfy all \(\text{AX}\) formulas. A state does not satisfy \(FS\) if it does not satisfy any of the sets contained in \(\text{SetOfFormulaSets}\).

We now show that given a formula set \(FS\), a call to \(\text{isConv}\) returns failure (an unsuccessful tableau) only after checking all possible sets that can be contained in \(\text{SetOfFormulaSets}\).

A call to \(\text{isConv}\) processes formulas as follows.

(i) A propositional formula \(p \in FS\) is always checked against the labels of the given state \(s\) in the parallel composition.

(ii) A negated proposition \(\neg p \in FS\) is always checked against the labels of the given state \(s\) in the parallel composition.

(iii) A conjunction is processed by adding both conjuncts to \(FS\), which are further processed depending on their type.

(iv) \(\text{AG}\) formulas are replaced by conjunctions which are processed further.

(v) \(\text{AU}\) formulas are replaced by disjunctions.

(vi) A disjunction \(\phi \lor \psi\) is processed as follows. First, \(\text{isConv}\) checks whether the current node can satisfy \(\phi\) by making a recursive call to \(\text{isConv}\) and passing \(FS\) in which the disjunction is replaced by the first disjunct \(\phi\). A failure (an unsuccessful tableau) is returned if an eventual subformula of \(\phi\) cannot be satisfied. During this call, \(\phi\) may be further broken down to subformulas propositions, negated propositions, disjunctions, and/or conjunctions which are handled depending on their respective types.

If the first recursive call returns failure, another call to \(\text{isConv}\) is made where \(FS\) is passed after the disjunction is replaced by \(\psi\). This call now checks all subformulas of \(\psi\) along with other formulas in \(FS\).

\(\text{isConv}\) returns failure when both recursive calls return failure, which happens after all different sets of commitments resulting from the disjunction have been checked. Note that during this process, it checks all possible sets of commitments resulting from the disjunction (along with other formulas in \(FS\)).

(vii) When \(FS\) contains only \(\text{AX}\) formulas (all present state commitments have been checked), \(\text{isConv}\) tries to find a subset of the set of successors of the current parallel composition state, such that each state in the subset satisfies all \(\text{AX}\) formulas. Note that \(\text{isConv}\) also ensures that the enabled subset ensures that the rules of the conversion refinement relation are satisfied.

\(\text{isConv}\) returns failure when no conforming subset that satisfies the above constraints could be found.

Given a set \(FS\) and the initial state \(s_{0,12}\) of the parallel composition, the call \(\text{isConv}\left(s_{0,12}, FS, \emptyset\right)\) results in each formula of \(FS\) being processed as discussed above. All different sets of formulas resulting from the presence of disjunctions are checked, including all future \(\text{AX}\) commitments that must be satisfied by the successors of the initial state.

For the call to \(\text{isConv}\) to return failure, no set of commitments present in the set of formula sets \(\text{extractSets}\) (FS) must be satisfied. In other words, there must be no possible converter states under which \(s_{0,12}\) satisfies \(FS\). However, as we are given that there exists a correct converter \(C\) with the initial state \(c_0\) such that \(c_0/s_{0,12} \models FS\), the statement that \(\text{isConv}\) may return FALSE_NODE cannot be true.

Hence, if there exists a converter that can guide a given parallel composition to satisfy a given set of commitments, \(\text{isConv}\) will never return FALSE_NODE. In other words, \(\text{isConv}\) will be always able to find a converter.

Proof. The proof of Theorem 1 follows from the above lemmas. □

ACKNOWLEDGMENTS

The authors acknowledge Dr. Alain Girault and Professor Zoran Salcic for their help in formulating this approach. The authors would also like to thank the anonymous reviewers of SLA++p’07 and Eurasip JES (Special Issue: SLA++p’07 and SLA++P’08) for their invaluable feedback.

REFERENCES


