Editorial

The aim of this workshop is to bring together researchers from academia and industry interested in formal modelling approaches as well as associated analysis and reasoning techniques with practical benefits for embedded software and component-based software engineering.

Component-based software design has received considerable attention in industry and academia since object-oriented software development approaches became popular. Recent years has seen the emergence of formal and informal techniques and technologies for the specification and implementation of component-based software architectures. With the growing need for safety-critical embedded software, this trend has been amplified.

Formal methods have sometimes not kept up with the increasing complexity of software. For instance, a range of new middleware platforms have been developed in both enterprise and embedded systems industries. Often, engineers use semi-formal notations such as UML 2.0 to model and organise components into architectures. FESCA aims to address the open question of how formal methods can be applied effectively to these new contexts.

For FESCA 2006, we have as invited speaker, Frantisek Plasil, Charles University, Prague, Czech Republic. We have accepted a total of 8 papers covering a broad range of topics within FESCA. Sessions are organised around topics and we hope to achieve interesting discussions after each session and throughout the day. All papers will appear in a final version of the proceedings published by Electronic Notes in Theoretical Computer Science, Elsevier B.V.

Juliana Bowles
Iman Poernomo
Ralf Reussner
Contents

Session - Component Models and Specification
DisCComp - A Formal Model for Distributed Concurrent Components
   A. Rausch
   1
Towards Multiple Access in Generic Component Architectures
   M. Klein, J. Padberg, and F. Orejas
   21
Extending a Component Specification Language with Time
   B. Metzler and H. Wehrheim
   43

Session - Component Contracts and Component Selection
Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation
   H. Koziolek and V. Firus
   65
A Contract-based Approach to Specifying and Verifying Safety Critical Systems
   W. Dong, Z. Chen and J. Wang
   85
Only the Best Can Make It: Optimal Component Selection
   L. Gesellensetter and S. Glesner
   100

Session - Model Checking
Local Module Checking for CTL Specifications
   S. Basu, P.S. Roop and R. Sinha
   122
Specification and Generation of Environment for Model Checking of Software Components
   P. Parizek and F. Plasil
   141
DisCComp — A Formal Model for Distributed Concurrent Components

Andreas Rausch ¹

Fachbereich für Informatik – AG Softwarearchitektur
Technische Universität Kaiserslautern
D-67653 Kaiserslautern, Germany

Abstract
Most large-scaled software systems are structured in distributed components to manage complexity and have to cope with concurrent executed threads. System decomposition and concurrent flow of execution are orthogonal. A sound semantic model that is powerful enough to handle distributed concurrent components but also realistic enough to provide a foundation for component technologies actually in use is still missing. Therefore, the paper introduces such an operational semantics for distributed concurrent component-based systems. Based on this formal model, UML-based modeling techniques are introduced. Tool support for modeling, code generation, and system execution is provided.

Key words: Software Architecture, Distributed Systems, Component-based Software, Concurrency, Operational Semantics, UML-based Description Techniques, Code Generation

1 Introduction
Software engineers are confronted with steadily increasing complexity of the software systems under development. On the other hand, we depend more and more on these software systems in our daily life. Hence, software engineers have to guarantee their dependability. Nevertheless, the development of software systems still includes a high level of uncertainty. More than 70 % of the development projects are not successful [1].

To deliver large-scaled software systems, like for instance enterprise applications, the software architecture is a key success factor. Its - usually hierarchical – decomposition into components is crucial for development and maintenance of software systems.

¹ Email: rausch@informatik.uni-kl.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Moreover, enterprise applications are usually interactive software systems and have to serve multiple users at the same time. For that reasons these kinds of software systems are distributed systems and have to serve concurrently multiple user requests. To sum up, these software systems have to cope with distributed concurrent components.

To implement those kinds of software systems with distributed concurrent components usually an object-oriented programming language like Java or C++ is used. Programming languages like Java offer basic constructs for concurrent programs, like for instance the util.concurrent library. Developing large-scaled distributed and concurrent systems based on these primitives is error-prone. Higher-level abstractions are needed. To cope with distribution and network communication, component technologies like CORBA, J2EE, and .NET are broadly used. These technologies basically provide a component model and a remote method call. Thereby they lift the structuring of systems from objects to components and the concept of a blocking synchronous method call within a single process to a distributed environment.

The provided component model of the underlying component technology is used for the hierarchical component structure and their distribution given by the software architecture. The remote method call is used to abstract from distribution and concurrent thread execution. Thereby concurrently executed threads “jump” from one possible remote component to another during system execution.

In fact threads cannot jump from one component to another remote component in case of a method call. However the used component technology simulates this behavior by blocking the calling thread in the calling component, starting a new thread in the remote component to process the requested method, and finally re-activating the blocked calling thread after the results of the remote computation are available. For programmers using those technologies it looks like a single thread jumping from one component to another. Hence the predominant programming model in distributed concurrent component systems is based on concurrent thread execution and distributed component structure which are orthogonal.

In practice, before programmers start coding different specification artifacts have to be created to model the system under development. The primary focus is on the description of the component structure of the system using description languages like the UML [2]. UML provides diagrams for static and dynamic behaviour. These techniques may be extended with elements describing more precise aspects of behaviour, such as JML [3] and OCL [4]. When it comes to integrate and refine these different parts of the models towards an implementation a clear semantical model of the relationship between the concurrent control flow and the component structure is missing, yet.

Various approaches have been elaborated to extend UML and to come up with a precise semantics for distributed and parallel systems [5,6,7]. All these approaches are based on active objects. Each concurrently executed
thread belongs exclusively to a single object. Thereby, the established model - concurrent threads jumping from one component resp. object to another - is simply ignored.

Moreover, various formal models for modelling concurrent systems have been elaborated over the last several decades: Although Hoares CSP approach is of general nature it does not incorporate modern component-based programming paradigms [9]. In the theoretical foundation of object-oriented languages provided by Martin Abadi and Luca Cardelli [10], in Focus [11], and in *-Calculus [12] concurrency is modelled in the sense of active objects resp. components or agents. But again, it does not support concurrent threads jumping from one object resp. component or agent to another.

Finally early approaches to integrate of the mentioned component technologies – CORBA, J2EE, and .NET - with UML have been developed, like for instance the UML/EJB mapping specification [8]. Again, a precise semantic mapping between the specification language and the programming platform is still a matter of research.

To sum up, there is still a large gap between the predominant programming model - concurrent threads jumping from one distributed component to another - and the existing formal as well as informal specification techniques. Even so, both are used by software engineers. Hence, they have to bridge the gap similar to a circus artist. This is a dangerous error prone task. Consequently, we need a sound semantic model that is powerful enough to handle concurrent distributed components but also realistic enough to provide a foundation for component and programming technologies actually in use.

The rest of the paper is structured as follows: The next section introduces a simple program to show the most important behavioral aspects of distributed concurrent component-based systems. In the following Sections 3 to 6, an operational semantics for distributed concurrent component-based software systems is elaborated. In Section 7 we show how this kind of distributed concurrent component-based software system can be modeled using UML. In Section 8 a short introduction into the modeling, code generation, and system execution tool environment for the presented approach is given. A short conclusion rounds the paper up.

2 Concurrent Program Sample

Assume a simple computer game board application, like a chess game. The human user enters his game move. The computer performs the corresponding changes on the game board data, then the computer calculates its move, and finally executes its move on the game board. While the computer calculates its move, the computer game usually allows the user to switch the game board side. Note that this is one reason why this simple application has to be concurrent. Whenever the computer is processing the move request of the
user, it still must react to side switch requests. Accepting user input during periods of prolonged processing time is a typical reason for interactive systems to be concurrent.

The Java program shown in Figure 1 is a concurrent program demonstrating the controller of such a computer game board application. The class SimulatedUser simulates the user. The move or the switch button is pressed randomly by the user. The operating system (class OperationSystem) receives either the move or the switch event. It creates a new thread to call the registered event handler of the application. Hence, a formal model for concurrent component-based systems has to support asynchronous message communication as well as concurrent method calls.

The class FourWins implements the controller of such a computer game board application. Instead of performing real moves on the game board our sample program prints ‘H’s and ‘C’s to illustrate whether a human move has been performed or a computer move. Repeated calling of gameMove() causes an initially alternating sequence of ‘H’s and ‘C’s. Whereas calling switchSide() may result in a ‘CC’ or a ‘HH’ on the standard output. This depends on whether the concurrent switchSide() is called before the computer has completed the calculation of his move (‘HH’) or thereafter (‘CC’).

However, concerning the concurrent game board control the simple Java program is fully featured. As one can see, managing the concurrency in the case of these two simple interleaving functions is a complex issue for a programmer. The concurrent threads share common variables. Hence, the proposed operational semantics for concurrent components has to support a shared global state.

Finally, the class FourWins could serve as an observable which can be observed for visualization or analyzing purpose. Other components may add themselves as observers. Thereby object instances may be created and deleted. Moreover the connections between these objects represented by references and pointers may be changed during system execution like for instance by adding a new observer to the list of observers within the class FourWins. Thus, the operational semantics introduced in the next section has to support those kinds of dynamically changing structures.

---

2 The observer mechanism is not shown in the code sample.
import java.util.Random;

public class SimulatedUser {
    public static void main(String[] args) {
        // initialize os
        OperationSystem theOS = new OperationSystem();

        // user can either make a game move or switch
        // the board side - randomized decided
        while (!(new Random().nextInt(100) == 50)) {
            if (new Random().nextInt(10) < 7) {
                // game moves are sequentialized; the corresponding
                // button is disabled until the previous move ends
                synchronized (FourWins.gameMoveButtonDisabeled) {
                    theOS.gameMoveEvent();
                }
            } else {
                theOS.switchSideEvent();
            }
        }
    }
}

public class OperationSystem {
    // initialize the only application of the os
    private FourWins theFourWinsApp = new FourWins();

    // for each received user event provide a thread in the
    // os and call the corresponding handler on the app
    public void gameMoveEvent() {
        new Thread(new Runnable() {
            public void run() {
                theFourWinsApp.gameMove();
            }
        }).start();
    }

    public void switchSideEvent() {
        new Thread(new Runnable() {
            public void run() {
                theFourWinsApp.switchSide();
            }
        }).start();
    }
}

public class FourWins {
    static public Boolean gameMoveButtonDisabeled = new Boolean(false);
    private boolean humanPlayerNext = true;
    private boolean computerPlayerIsCalculating = false;

    public void gameMove() {
        // disable game move button - user cannot perform
        // another move until this move has been finished
        synchronized (gameMoveButtonDisabeled) {
            // human game move is simulated
            synchronized (FourWins.class) {
                if (humanPlayerNext) {
                    System.out.print("H");
                    humanPlayerNext = false;
                    computerPlayerIsCalculating = true;
                }
            }
            // simulating computer calculating period to give the human the
            // chance to switch board side resulting in an double H output
            try {
                Thread.sleep(0,1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // computer game move is simulated
            synchronized (FourWins.class) {
                computerPlayerIsCalculating = false;
                if (!humanPlayerNext) {
                    System.out.print("C");
                    humanPlayerNext = true;
                }
            }
        }
    }

    public void switchSide() {
        // either computer or human game move is simulated
        // depending on the actual active move in gameMove()
        synchronized (FourWins.class) {
            if (!computerPlayerIsCalculating) {
                System.out.print("C");
            } else {
                humanPlayerNext = true;
            }
        }
    }
}

Fig. 1. Concurrent program sample
3 Basic Concepts

This section elaborates the basic concepts of the proposed formal model for distributed concurrent component-based software systems. Such a model incorporates two levels: The instance level and the description level [13].

The description level - described in Section 7 - contains a normalized abstract description of a subset of common instance level elements with similar properties. The instance level - described in the Sections 3 to 6 - is the reliable semantic foundation of the description level. It provides an operational semantics for distributed concurrent components - it is an abstraction of existing programming models like CORBA, J2EE, and .NET. Thereby, it defines the universe of all possible software systems that may be specified at the description level and implemented using the mentioned programming models.

The instance level of our proposed formal model for distributed concurrent components must be powerful enough to handle the most difficult behavioral aspects as presented in the previous section:

- dynamically changing structures,
- shared global state,
- asynchronous message communication, and
- concurrent method calls.

Figure 2 summarizes these behavioral aspects of the formal model for distributed concurrent components at the instance level on an abstract level. Thereby, software systems consist of a set of disjoint instances during runtime: system, component, interface, attribute, connection, message, thread, and value. In order to uniquely address these basic elements of the instance level we introduce the infinite set \( \text{INSTANCE} \) of all instances:
The presented four behavioral aspects of distributed concurrent component-based systems are described in the following.

3.1 Structural Behavior

A system may change its structure dynamically. Some instances may be created or deleted (ALIVE). New attributes resp. interfaces may be assigned to interfaces resp. components (ALLOCATION resp. ASSIGNMENT). Interfaces may have a directed connection to other interfaces (CONNECTS):

\[
\begin{align*}
&\text{ALIVE} = \text{def } \text{INSTANCE} \rightarrow \text{BOOLEAN} \\
&\text{ASSIGNMENT} = \text{def } \text{INTERFACE} \rightarrow \text{COMPONENT} \\
&\text{ALLOCATION} = \text{def } \text{ATTRIBUTE} \rightarrow \text{INTERFACE} \\
&\text{CONNECTS} = \text{def } \text{CONNECTION} \rightarrow \{ \{\text{from}, \text{to}\} \mid \text{from}, \text{to} \in \text{INTERFACE}\}
\end{align*}
\]

3.2 Valuation Behavior

A system’s state space is not only determined by its current structure but also by the values of the component’s attributes. Mappings of attributes or parameters to values of appropriate type are covered by the following definition:

\[
\text{VALUATION} = \text{def } \text{ATTRIBUTE} \rightarrow \text{VALUE}
\]

3.3 Communication Behavior

Sequences of messages represent the fundamental units of asynchronous communication. In order to model message-based asynchronous communication, we denote the set of arbitrary finite message sequences with MESSAGE*. Within each observation point components process message sequences arriving at their interfaces and send message sequences to other interfaces:

\[
\text{EVALUATION} = \text{def } \text{INTERFACE} \rightarrow \text{MESSAGE}^*
\]

3.4 Execution Behavior

Besides asynchronous communication, synchronous method calls performed by concurrent executed threads is the predominant execution mechanism in contemporary software systems. Each method is called at a certain interface (CALL). Hence, to model a thread’s call stack, we denote the set of arbitrary finite method call sequences with (INTERFACE × CALL)*. Each thread has its own method call history - its call stack (EXECUTION). Note that threads may change the hosting component in case of a method call at an interface belonging to another component:

\[
\text{EXECUTION} = \text{def } \text{THREAD} \rightarrow (\text{INTERFACE} \times \text{CALL})^*
\]
3.5 System Snapshot

Based on the former definitions, we are now able to characterize a snapshot of a software system. Such a snapshot captures the current structure, variable valuation, actual received messages, and current method calls. Let \( \text{SNAPSHOT} \) denote the type of all possible system snapshots:

\[
\text{SNAPSHOT} = \text{def} \ ALIVE \times \text{ASSIGNMENT} \times \text{ALLOCATION} \times \text{CONNECTS} \\
\times \text{VALUATION} \times \text{EVALUATION} \times \text{EXECUTION}
\]

4 System Behavior

In contrast to related approaches like [11], we do not focus on timed streams but on execution streams. We regard observation points as an infinite chain of execution intervals of various lengths. Whenever a thread’s call stack changes - in case of a new method call or a method return - a new observation point is reached. We use the set of natural numbers \( \mathbb{N} \) as an abstract axis of those observation points, and denote it by \( E \) for clarity.

Furthermore, we assume an observation synchronous model because of the resulting simplicity and generality. This means that there is a global order of all observation points and thereby of all method calls and returns. Note that this is not a critical constraint. Existing distributed component environments like CORBA, J2EE, and .NET control and manage all method calls and returns. Such a component environment may transparently force a global order of all method calls and returns.

We use execution streams, i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over observation points. An execution stream - more precisely, a stream with discrete execution interval - of elements from the set \( X \) is an element of the type

\[
X^E = \text{def} \ \mathbb{N}^+ \rightarrow X, \text{ where } \mathbb{N}^+ = \text{def} \ \mathbb{N} \setminus \{0\}
\]

Thus, an execution stream maps each observation point to an element of \( X \). The notation \( x^e \) is used to denote the element of the valuation \( x \in X^E \) at the observation point \( e \in E \) with \( x^e = x(e) \).

Execution streams may be used to model the behavior of software systems. Accordingly, \( \text{SNAPSHOT}^E \) is the type of all system snapshot histories or simply the type of the behavior relation of all possible software systems:

\[
\text{SNAPSHOT}^E = \text{def} \ ALIVE^E \times \text{ASSIGNMENT}^E \times \text{ALLOCATION}^E \times \text{CONNECTS}^E \\
\times \text{VALUATION}^E \times \text{EVALUATION}^E \times \text{EXECUTION}^E
\]

Let \( \text{Snapshot}^E_s \subseteq \text{SNAPSHOT}^E \) be the behavior relation of an arbitrary system \( s \in \text{SYSTEM} \). A given snapshot history \( \text{snapshot}_s \in \text{Snapshot}^E_s \) is an execution stream of tuples that capture the changing snapshots \( \text{snapshot}_s^e \) over

\[\text{\textsuperscript{3}}\text{ In the remainder of this paper we will use this shortcut. Whenever we want to assign a relation } X (\text{element } x) \text{ to a system } s \in \text{SYSTEM we say } X_s(x_s).\]
observation points $e \in E$.

Obviously, a couple of consistency conditions can be defined on a formal behavior $\text{Snapshot}_E^e \subseteq \text{SNAPSHOT}_E$. For instance, it may be required that all attributes obtain the same activation state as the interface they belong to:

$$\forall a \in \text{Attribute}_s, i \in \text{Interface}_s, e \in E. \text{allocation}_s^e(a) = i \Rightarrow \text{alive}_s^e(a) = \text{alive}_s^e(i)$$

Or furthermore, instances that are deleted are not allowed to be reactivated:

$$\forall i \in \text{Instance}_s, e, n, m \in E. e < n < m \land \text{alive}_s^e(i) \land \neg \text{alive}_n^s(i) \Rightarrow \neg \text{alive}_m^s(i)$$

We can imagine plenty of those consistency conditions. A full treatment is beyond the scope of this paper, as the resulting formulae are rather lengthy. A deeper discussion of this issue can be found in [14,15].

5 Thread Behavior

A system’s observable behavior is a result of the composition of all thread behaviors. To show this coherence, we first have to provide the behavior formalization of a single thread. In practice, transition relations are an adequate behavior representation. In our formal model we use a novel kind of transition relation: in contrast to a ‘normal’ transition relation - a relation between a state and its successor state - the presented transition relation is a relation between a certain part of the system-wide current snapshot and a certain part of the threads’ wished system-wide successor snapshot after performing a method call or return:

$$\text{BEHAVIOR} = \text{def} \ \text{SNAPSHOT} \rightarrow \text{SNAPSHOT}$$

Let $\text{behavior}_t \subseteq \text{BEHAVIOR}$ be the behavior of a thread $t \in \text{THREAD}_s$ in the system $s \in \text{SYSTEM}$. The informal meaning of the thread behavior is as follows: Each thread performs a sequence of operations represented by transition relations. Each operation resp. transition relation $\text{transition} \in \text{behavior}_t$ can intuitively be seen as an atomic piece of program code, which has the following structure:

(i) The thread evaluates the part of the system-wide snapshot, which is relevant of its execution. If this part of the system-wide snapshot fits, given by the first snapshot of the tuple $\text{transition}$,

(ii) the thread requests a set of changes on the system-wide snapshot. Thus, the thread wants the system to be consistent with the system-wide successor snapshot in the next step given by the second snapshot of the tuple $\text{transition}$.

(iii) Finally, the thread wants to perform a new method call or return. Again this is given by a call-stack change described in the function $\text{execution}_t \subseteq \text{EXECUTION}$, which is part of the second snapshot in the tuple $\text{transition}$.

Note that the behavior relation of threads is a function, not a relation. Thus, non-determinism cannot be expressed. To represent non-determinism under specification of thread’s behavior could especially be a probate solution. However this is not a general restriction of the proposed approach.
6 Behavior Composition

Consequently, we need some specialized run-time system that asks all threads - one by one - if one wants to perform a new method call or return from a method call. Whenever a thread wants to perform a new method call or return, which means that its behavior relation fires, the run-time system composes a new well-defined system-wide successor snapshot based on the thread’s requested changes and the current system-wide snapshot.

Hence, such a run-time system is similar to a virtual machine. It observes and manages the execution of all threads. Again, this is not a critical constraint even in a concurrent and distributed environment. Existing distributed component environments like CORBA, J2EE, and .NET control and manage all executed components within the environment. In Section 8 we show how the proposed approach can be implemented by extending such an existing and widely used component environment like CORBA, J2EE or .NET.

To sum up, the main task of such a run-time system is to determine the next system snapshot $\text{snapshot}_{e+1}^s$ from the current snapshot $\text{snapshot}_e^s \in \text{Snapshot}_e^s$. In essence, we can provide formulae to calculate the system behavior from the initial configuration $\text{snapshot}_0^s$, the behavior relations $\{\text{behavior}_{t_1}, ..., \text{behavior}_{t_n}\}$ of all threads $t_1, ..., t_n \in \text{THREAD}_s$, $n \in \mathbb{N}$, and external stimulations via asynchronous messages and synchronous method calls at free interfaces. Note that free interfaces are interfaces that are not connected with other interfaces and thus can be stimulated from the environment.

Before we can come up with the final formulae to specify the run-time system, we need a new operator on relations. This operator takes a relation $X$ and replaces all tuples of $X$ with tuples of $Y$ if the first element of both tuples is equal 4:

$$X \triangleleft Y = \text{def} \{ a | a \in Y \land (a \in X \land \pi_1(\{a\}) \cap \pi_1(Y) = \emptyset) \}$$

We are now able to provide the complete formulae to determine the next system snapshot $\text{snapshot}_{e+1}^s$:

next-snapshot: $\text{SNAPSHOT} \rightarrow \text{SNAPSHOT}$

$$\text{next-snapshot}(\text{snapshot}_e^s) = \text{def} \text{snapshot}_{e+1}^s =$$

$$= (\text{alive}_{e+1}, \text{assignment}_{e+1}, \text{allocation}_{e+1}, \text{connects}_{e+1}, \text{valuation}_{e+1}, \text{evaluation}_{e+1}, \text{execution}_{e+1})$$

$$\text{alive}_{e+1} = \text{alive}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{assignment}_{e+1} = \text{assignment}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{allocation}_{e+1} = \text{allocation}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{connects}_{e+1} = \text{connects}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{valuation}_{e+1} = \text{valuation}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{evaluation}_{e+1} = \text{evaluation}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

$$\text{execution}_{e+1} = \text{execution}_{e}'(\text{behavior}_{\text{thread} \rightarrow \text{next}}(\text{snapshot}_e^s)) \land$$

---

4 Note that the “standard” notation $\pi_{i_1,...,i_n}(R)$ denotes the set of $n$-tuples with $n \in \mathbb{N}$ and $n \leq r$ as a result of the projection on the relation $R$. Whereas in each tuple in $\pi_{i_1,...,i_n}(R)$ contains the elements at the position $i_1, ..., i_n$ of the corresponding tuple from $R$ with $1 \leq i_k \leq r$, with $k \in \{1,...,n\} \subseteq \mathbb{N}$. 

10
valuation$^{s+1}_{s} = valuation_{s \triangleright_{\pi}(behavior_{next\_thread}(snapshot_{s}))}$ \land \\
evaluation_{s} = \pi_{6}(behavior_{next\_thread}(snapshot_{s})) \land \\
execution_{s} = execution_{s \triangleright_{\pi}(behavior_{next\_thread}(snapshot_{s}))} a_{=7}(message\_execution(snapshot_{s}))$

Intuitively spoken, the next system snapshot $\text{snapshot}_{s}^{+1}$ is a tuple. Each element of this tuple, for instance $\text{assignment}_{s}^{+1}$, is a function that is determined simply by merging the former function $\text{assignment}_{s}$ and the ‘delta-function’ of $\pi_{2}(behavior_{next\_thread}(snapshot_{s}))$. This ‘delta-function’ includes all ‘wishes’ of the next relevant thread determined by the function $\text{next\_thread}$.

This intuitive understanding does not completely hold for $\text{alive}_{s}^{+1}$, $\text{evaluation}_{s}^{+1}$ and $\text{execution}_{s}^{+1}$. In $\text{alive}_{s}^{+1}$ and $\text{execution}_{s}^{+1}$, not only the wishes of thread $\text{next\_thread}$ have to be included. These wishes must contain the thread’s actual method call or return. Additionally they may contain new parallel threads created by the current thread.

Moreover, $\text{alive}_{s}^{+1}$ and $\text{execution}_{s}^{+1}$ also contain the result of the application of the function $\text{message\_execution}(\text{snapshot}_{s})$. This function includes new threads created to process the asynchronous messages. Thereby, for each asynchronous message - given by $\text{evaluation}_{s}$ which is included in $\text{snapshot}_{s}$ - a new thread is created in $\text{alive}_{s}^{+1}$ to execute the corresponding request in $\text{execution}_{s}^{+1}$.

The intuitive understanding for $\text{message\_execution}$ is defined as follows:

$\text{message\_execution}(\text{SNAPSHOT}) \rightarrow \text{SNAPSHOT}$

$\forall i \in \text{Interface}_{s}, m \in \text{Message}_{s}, m \in \text{evaluation}_{s}(i) \Leftrightarrow \exists t' \in \text{Thread}_{s}, \neg \text{alive}_{s}(t') \land \text{alive}'(t') \land \text{execution}'(t') = \{(i, m)\}$

Intuitively spoken for each asynchronous message a new thread is activated and the corresponding call stack is initialized. As all asynchronous messages are with each observation point transformed to corresponding concurrently executed threads, the new system snapshot has only to contain the new asynchronous messages, as denoted by $\text{evaluation}_{s}^{+1} = \pi_{6}(behavior_{i}(\text{snapshot}_{s}))$.

Note that thereby, the delivery of asynchronous message takes some time, exactly one observation point. To model network latency or network failure one would have to provide a more sophisticated function $\text{message\_execution}$. Thus, not only delay and loss of asynchronous messages could be integrated but also network related failures in executing method calls.

To complete the formal model, the function $\text{next\_thread}$ has to be defined:

$\text{next\_thread} : \rightarrow \text{THREAD}$

This function returns the next thread to be visited by the run-time system. To provide a simple but general model we propose a round-robin model. Therefore, a given strict order of all active system’s threads is required. $\text{next\_thread}$ follows this given order and provides the next relevant thread to be visited and integrated into the system-wide snapshot by the run-time system.

Note that one can integrate additional features into the model providing other implementations of the function $\text{next\_thread}$, like for instance non-
determinism and priority-based thread scheduling. Non-determinism could be used to model an unsure execution order or to support under-specification.

Whenever concurrent threads or components are executed, inconsistency or deadlocks may occur. A deadlock concerning elements explicitly modeled in the semantics - like for instance two threads each locking an attribute and waiting to get the lock on the other’s thread attribute - cannot occur in this model as all threads are visited one by one and each thread has to release all blocked resources after it has been visited. However, deadlocks on a higher level, like for instance one thread waits for a given condition to become true and another thread waits for this thread to make another condition true, can not be detected in advance.

The model does not suppress inconsistent situations but it helps to detect them. In order to ensure that the next system snapshot $\text{snapshot}^{e+1}$ is well-defined, a single basic condition must be satisfied: all elements in the wished successor snapshot given by $\text{behavior}_{\text{next_thread}}(\text{snapshot}^e)$ that cause a change in the resulting next system snapshot must not be changed after the thread $\text{next_thread}$ has made his last method call or method return.

For instance, assume that a thread performs a method call. The value of an attribute is 5 as the thread has started the method call execution and the thread wants to change the value to 7 as it returns from this method call. At the observation point where the thread returns from the method call the value of the attribute is already 6, as another thread has changed the value in the meantime. Hence, a possible inconsistency caused by concurrent thread execution occurs.

A run-time system implementing the function $\text{next_snapshot}$ has to calculate the next system snapshot. Thereby it can observe this consistency predicate and verify whether such a possible inconsistency situation occurs or not. If the run-time system detects such a possible inconsistent situation it may stop the system execution for reliability reasons. Note that this formal consistency concept for concurrent threads is similar to optimistic locking techniques in databases.

7 Description Technique

The operational semantics presented in the previous sections represents the instance level. It defines our understanding of distributed concurrent component-based software systems. Based on the operational semantics we can provide run-time consistency checks as discussed in the previous section. Moreover the operational semantics is the semantic foundation of the description level.

The description level contains a set of proper specification and modeling techniques to elaborate and specify distributed concurrent components. Once the description level is formally founded based on the operational semantics, we may generate executable code out of the specifications which are then executed within the run-time environment (cf. Section 8).
The presented description technique is based on UML. Consider the well-known observer pattern [16]. Figure 3 shows a simple component-based version of the observer pattern. The component Data represents the observable object of the observer pattern. It provides exactly one Observable interface. This interface encapsulates the state to be observed represented by the attribute sState. Whenever the state of sState is changed, the method notify() is called. This results in sending an asynchronous message update() to all interfaces Observer which are connected via a connection Observation. Additionally to the normal observable functionality, the interface Observable provides the method deploy(). Calling deploy() results in creating a new component View with a corresponding interface Observer and attaching the interface to the Observable interface of the called component Data.

In order to specify the behavior of methods and message processing for each method and for each message, an UML activity diagram is provided.
Here, we use an extended version of UML activity diagrams or alternatively a textual representation of those kinds of activity diagrams.

Figure 4 shows the textual and Figure 5 the graphical variant of the UML activity diagram describing the behavior of the method notify(). Thereby, first a sequence of the connected Observer interfaces is requested. Then, to each Observer interface in this sequence the asynchronous message notify() is sent.

All first class elements of the operational semantics are explicitly represented in the description technique, like components, interfaces, connections, attributes, messages, and methods. Moreover, for each primitive operation in the operational semantics a specific textual and graphical construct is available.

For instance, Figure 6 shows the textual description of the method deploy(). As already mentioned, calling this method results in creating a new component View with a corresponding interface Observer and attaching the interface as new observer. Within the description of this, method specific syntactical constructs for creating components, interfaces, and connections are provided. The proposed description technique contains specific syntactical constructs for all first class elements and for all primitive operations. Except for threads, no specific representation is provided. As our operational semantics manages the integration of concurrent executed threads within a component-based system, there is no need for handling concurrency issues, like for instance synchronized statements, semaphores, and monitors, on the specification level. Hence
the error prone task of programmers to synchronize various parts of the code of components as shown in Figure 1 is no longer needed. Especially in a component-based programming model it is very important to avoid those kind of fine-grained code synchronization used in Figure 1 as it damages component encapsulation.

Figure 7 illustrates an extended UML sequence diagram. This sequence diagram shows an observer component attached to an observable component. First the message `deploy()` is called. A new observer is created and attached to the observable. Then the state of the observable is changed. This results in calling `notify()`, which leads to sending asynchronous messages `update()` to all attached observers. Once the messages are processed, the observers request the new state from the observable component. Figure 7 illustrates a possible thread structure and concurrency situation. Each thread is shown by a grey box. The communication sequences shown within these boxes are performed by the corresponding thread.

Thread 1 and Thread 2 could be the same thread. This depends on the caller. Either both have the same caller, then Thread 1 and Thread 2 are identical, or the callers are different, then Thread 1 and Thread 2 are not identical. Moreover Thread 3 and Thread 4 are newly created threads to process the received asynchronous messages `update()`.

In addition, Figure 7 illustrates that the sequence is split up into several observation points applying our operational semantics. These observation points are the points in execution to synchronize all concurrently executed threads. As discussed in Section 4, an observation point exists for each method call or method return following our operational semantics. Consequently, for each thread entering as well as leaving the communication sequence an observation point exists. Moreover, as shown in Thread 2, threads may be divided into several observation points. Either as a method call is performed within the thread - in our case the method `notify()` is called - or another thread causes an observation point - in our case Thread 3 causes two additional observation points within the execution of Thread 2.
8 Tool Support

In order to apply the presented concepts in practice, proper tool support is a key success factor. As already mentioned, we have developed a tool support for modeling, code generation and system execution - called DesignIt [17].

Software engineers can use DesignIt to model the component-based software system. To do this, the software engineer uses a CASE tool to develop a UML-based model of the desired component-based software systems. In doing so, the description techniques presented in the previous section should be used - component diagrams for the static structure and syntax, and activity diagrams to model the behavior of methods and messages.

Currently, the CASE tool Together is supported directly. All modeling samples, the UML profile defining UML extensions, and the XML generator are implemented as plug-ins for Together. However, any other CASE tool may be used. For this, a corresponding XML generator has to be provided.
As shown in Figure 8, a XML file is generated out of the UML model (step 1). This XML file is a simple textual representation of the UML model. We could have used XMI as textual representation. XMI has to support the complete bunch of UML elements. We only need the proposed extended component and activity diagrams for the DesignIt environment. To keep the following processing tasks simple, we have decided to define our own simple XML representation with respect to the proposed extended component and activity diagrams.

XSLT transformations are used to generate the complete Java program code out of the XML files (step 2). The resulting Java components are then executed and debugged within the run-time system (step 3). When defects are detected in this step, the program can be debugged and analyzed.

Once the problem is identified, it can be solved by changing the UML model (step 4). Afterwards, the whole cycle can be iteratively applied until the implementation is correct. To sum up, the DesignIt tool environment supports pure forward code generation. As the code is generated completely - no more coding is required - there is no need for backward code engineering.

The execution and debugging environment of DesignIt is distributed itself. It is implemented using CORBA as distribution and network communication technique. The execution environment is a CORBA server. For each component type, a separate CORBA server is started. Even the debugger is started within an own CORBA server. Hence, the execution environment, the debugger, and each component type can be executed on a separate computer.

Figure 9 shows a screenshot of the debugging environment of DesignIt. On the top level of the tree, each available server is shown. In the shown case, three additional distributed servers exist: one for the runtime environment, one for all instances of component type CB, and the other for all instances of
Moreover, for each component instance the current status and its syntactical interface are shown.

Using this debugging environment, asynchronous messages or method calls from outside the system can be initiated. Once the messages resp. method calls are stored in the run-time environment, the whole distributed and concurrent system can be executed step by step. Each step represents an observation point in the sense of our operational semantics. Once an inconsistency appears, the debugging environment stops the system execution and informs the software engineer about the presence of a possible inconsistency situation as discussed in Section 6. The software engineer then can analyze the situation and fix the model until it is correct.

9 Conclusion

The ability to develop and maintain distributed concurrent component-based software systems is essential for modern software engineering. To bridge the gap between the vertical component-based decomposition of software systems and the horizontal concurrent execution flow in software systems, an operational semantic for distributed concurrent components has been elaborated. This model also includes hierarchical components - software systems that contain components which are again composed out of so-called sub-components. This has not been presented in the paper but it is included in the complete formal semantics in [15].

This model provides a sound and realistic semantic foundation for this kind of software systems: it is powerful enough to handle dynamically changing structures, shared global state, asynchronous message communication, and
concurrent method execution. The overall system behavior can be calculated from the concurrently executed threads and their behavior relations. Based on the operational semantics inconsistent system states, especially caused by the concurrent execution, can be detected during run-time and further system execution can be stopped.

Moreover, textual and graphical description techniques have been presented to describe this kind of concurrent component-based software systems. A complete and formally founded semantically mapping of the description techniques to the operational semantics has not been presented in the paper. This has already been elaborated in [15].

Based on this semantics complete code generation and execution within the run-time environment is supported. Tool support for modeling, code generation and system execution has been implemented and used in small case studies. However, further improvement has to be done.

Moreover tool support for reasoning on the specifications could be addressed. Currently in [15] concepts for reasoning about the changes of component composition in case of component evolution are already elaborated and implemented. Further tool support concerning the consistency between specification and code may be a worth full improvement.

Case studies and industrial experiences have to be undertaken to show whether the proposed approach is of practical relevance or not. The formal model could be extended to integrate the concept of exceptions. Moreover, additional description techniques like for instance sequence diagrams to model system traces and test cases could be integrated. The tool support may be extended to integrated run-time test and verification, failure analysis assistance, and code generation for additional target platforms.

10 Acknowledgements

My thanks to Christian Bartelt, Arnd Poetzsch-Heffter, Marcus Reitz, Marcus Seiler, Thomas Ternit for input and comments to this work.

References


Towards Multiple Access in Generic Component Architectures

M. Klein, J. Padberg

Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany

F. Orejas

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract

The paper introduces an abstract framework for the specification of components with multiple require and provide interfaces that allows the specification of multiple access to a single provide interface. This framework can be regarded as a generalization of abstract hierarchical and connector-based component specification approaches. The main ideas are clarified in a sample specification, a component architecture for a web browser suite. For this, elementary nets are applied and are shown to be an instantiation of the abstract framework.

Key words: Component Architectures, Reduction Semantics

1 Introduction

By now component-based software development is becoming nearly a standard in large scale software engineering (see e.g. [21,22,33,34]), for several reasons: For example, components implemented once can easily be integrated in other projects requiring the same functionalities. It is possible to buy components with explicitly defined interfaces, thus, time pressure in the development of software projects can be relaxed by paying for a piece of code that is quickly integrated into the project. Changes of a component body, or even a full exchange, can be processed encapsulated, i.e. with no effects for the component’s environment as long as the corresponding interfaces are preserved. See [24] for a survey of component-based software engineering.

Acknowledgment: This work is partially supported by the TMR network SEGRAVIS.

Email: [klein, padberg]@cs.tu-berlin.de

Email: orejas@lsi.upc.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
But despite the wide acceptance of component-based software development approaches, there is still a lack of specification techniques suited for component-based design. Especially the application of formal specification techniques is hardly supported in a continuous fashion that comprises components and composition as well as the architecture. But whenever it is important to verify or to check the correctness of an implementation with respect to a specification, e.g. in the case of security relevant software, formal techniques equipped with a mathematical semantics have strong advantages compared to less formal techniques.

In [20] D. Garlan lists convincing arguments for the use of formal techniques architecture description languages. In [31] a survey (in German) over the use of formal techniques for the description of software architectures is given. Exemplarily stated there are: Process algebras are used for various architecture description languages, e.g. DARWIN [27,28], WRIGHT [2,3] or AEMILIA [4], but none of these specifies the component itself. SARA [17] is an early architecture description language using Petri nets for the description of the operational behavior. In [9,10] dualistic Petri nets are proposed. These describe the architecture using abstract representations of parallel process objects. In ZCL [11] is based on Z [36] a set-theoretic specification language. Z schemes are used to describe the architecture structure as well as the dynamic changes. COMMUNITY [18] and COOL [23] are architecture description languages that are founded on graph transformations. But in COOL no explicit component specification is given.

But there are only a few other approaches with the aim to combine component-based architectures and formal specification techniques in order to have a continuous formal technique. Those examine only a particular specification technique in contrast to our generic approach. E.g. [7] uses the integrated formal specification technique Korrigan to specify components and their composition. COMMUNITY [8,19] is a prototype language for architectural modeling that is founded on graph transformations.

In [13] a transformation-based hierarchical component concept that is generic with respect to the used specification technique and the applied transformation notion, has been presented. This first step to close the gap between formal specification techniques and real life component architectures has been followed by another concept using generic specifications and transformations. In [16,12] connector-based architectures have been introduced to enable the specification of components with multiple provide interfaces that are coordinated by connectors with several require specifications. Both approaches have been successfully instantiated to a variety of specification techniques [14,15]. In this paper a new, even more general approach is introduced which allows the specification of components with several require and provide interfaces, where require interfaces correspond to import interfaces and provide interfaces correspond to export interfaces in the previous transformation-based approaches.
Even more important, the approach allows the connection of different requirement specifications of one or more components with the same provide specification. This is a very common scenario in a software developing process. Fig. 1 shows a small architecture containing a web browser, a mail program, a text editor, and a file browser that is accessed by both, the web browser and the mail program. In Sect. 2 it is shown that both accesses operate on the same provide specification and how this is handled within the new approach.

As the above mentioned generic transformation-based approaches, this paper concentrates on a static view of architectures. See Sect. 5 for a discussion of possible extensions handling dynamic architectures. Since the main motivation for a formal approach to component architectures is to enable verification and model checking, it is necessary to calculate the common specification for a given set of component specifications, i.e. the given architecture. This process, explained in detail in Sect. 3, is suitable for many specification techniques and application scenarios.

2 Example: Web Browser Suite

In this section the sample architecture shown in Fig. 1 is explained in detail. We use elementary nets as specification technique. In Sect. 4 we sketch the formal instantiation of our generic framework to this technique.

\[ \text{COMP}_{WB} \quad \text{COMP}_{M} \quad \text{COMP}_{FB} \quad \text{COMP}_{TE} \]

Fig. 2. Architecture Graph of Web Browser Suite

Each component, in general and in the example, consists of a body and a set of provide and require specifications. E.g. the component web browser is given by \( \text{COMP}_{WB} = (\text{REQ}_{WB} \rightarrow \text{BOD}_{WB} \Leftarrow \text{PRV}_{WB}) \) where \( \text{REQ}_{WB} \) specifies the require interface, \( \text{BOD}_{WB} \) the body and \( \text{PRV}_{WB} \) the provide interface. Each of these specifications is given an elementary net. Fig. 2 shows the architecture graph that is the components and their connection.
The more detailed illustration in Fig. 7 shows all specifications of the example’s components and all connecting transformations and embeddings. The component index set of the architecture is given by $I = \{ WB, M, FB, TE \}$, which is the set of abbreviations for the component names: Web Browser, Mail, File Browser, Text Editor.

![Fig. 3. Web Browser Component](image)

The specification of the web browser component as shown in Fig. 3 contains two provide and a single require interface. The provide interfaces of the web browser component state that this component initially is in a state called idle and two different events can occur: the simple loading of a web page (load page) or loading and storing a file to disk. Both events lead to the same state, the idle state.

The provide specification nets are refined by the component’s body net. The place idle (expressing the initial state of the component) and the load page transition remain unchanged. The download_and_store transition is replaced by a subnet containing the three places controld, wait and file, and the transitions store_file, download! and browse_file. This subnet models that after each occurrence of the download! event the user has to start a file browser to determine the storage area and the save name of the file. After this selection, the actual download and the file saving are executed. The place controld ensures that the selected file is the result of the started browsing process. This file browsing process and the related places are in the component’s only require specification.

The component COMP_WB also contains the connections between its provide and require interfaces and the body. For the case of the require interfaces, the corresponding embeddings are quite obvious, and thus omitted. In Fig. 4 the provide interface and the body of component COMP_WB are connected. The only place of the provide interfaces is mapped to the same place in the body net. Since the transition load page remains unchanged in the body, it is
mapped to the subnet containing the transition `load_page` and the only place connected to the transition. The transition `download_and_store` is mapped to a net containing the whole body except the transition `load_page`.

The specification of the mailer component is depicted in Fig. 5. Besides a body net it contains a single provide interface and two require interfaces. Initially the provide interface of this component allows two events: `read_mails` and `write_mail`. After a mail has been written, the net is enabled to send this message. Both, the `send_mail` and the `read_mails` transitions lead to the initial
In the body net of the component the process of writing a mail is refined. First, the mailer starts a text editor which is then used to write the content of the email. Afterwards, the user is enabled to send the mail or to attach a file to it. The latter includes the browsing of a suiting file. Both, the creation of the email content and the browsing of an attachment, are to be provided by the component’s environment. This is expressed by the occurrence of the two transitions in the require interfaces.

Fig. 6 shows the components file browser and text editor. Both do not contain a require interface. The simple file browser is specified by only two transitions that offer to change the directory and to select finally the file. The text editor body is specified by three transitions, expressing the possibilities of writing a character, deleting a character and to finally export the written text.

Fig. 7 illustrates the architecture diagram of the web browser suite. It involves all specifications, transformations, and embeddings, but it disregards all the specific elementary nets given in Figures 3-6. The abstraction of this diagram is the architecture graph in Fig. 2.

In this sample architecture all but one connecting transformation are iden-
tity transformations, i.e. we have equality of the corresponding require and provide interfaces. The transformation $con_{M, rm2, TE}$ shown in Fig. 8 is different, since it actually applies the possibility to rename places along transformations of marked elementary nets. In general, our framework offers the possibility to connect require and provide interfaces by refining transformations. For the case of elementary nets this includes the possibility to replace transitions by subnets.

Since the main motivation for applying formal techniques to software engineering is verification we need to construct from the given components a single specification that can be verified with the corresponding tools of that specification technique. In the next section we define how components can be composed to larger ones. In the case of our sample architecture, the repeated application of the composition operation yields a component that contains the whole behavior of the browser suite. Fig. 9 shows the body of the resulting component.

After having derived the complete specification, we now can start verification or model checking, respectively, with respect to a given requirement specification, but this is not within the scope of this paper.
3 The Generic Framework

One central aim of this work is to define generic notions of components and composition operations capable of handling multiple access scenarios, as shown in the example of the previous section. As the approaches [13] and [16,12] this work applies generic specifications, embeddings and transformations to form components. Since not all classes of embeddings and transformations are suitable for this purpose we have to state some general requirements first. The validity of these requirements needs to be proven in the concrete specification technique when instantiating the generic concept.

3.1 General Requirements

Our generic technique requires a defined class of specifications, corresponding transformations and embeddings. Since the transformations are used in the framework to establish the connection between provide interfaces and the actual component specification, the component body, it is sensible to assume that the transformations define a class of refinements for the specifications. Since there exist so many notions of refinement, even for single specification techniques, this assumption should not be further formalized at the abstract level - but it has to be clarified when the concept is instantiated. In Sect. 2 we applied a refinement notion for elementary nets that allows mapping single transitions to whole subnets (see Sect. 4 for details).

For both, the transformations and the embeddings, we require a composition operation and a special identity instance. Moreover, it is necessary that the class of embeddings defines a subclass of the transformations, i.e. we require a mapping \( \text{trafo} : \text{EMB} \rightarrow \text{TRAFO} \) that selects a transformation for each embedding.

The extension property defined below is well-known from [12] and [13]. It states that a single transition can be applied to a larger context.

**Definition 3.1 Extension Property** Given an embedding \( e : \text{SPEC}_R \rightarrow \text{SPEC} \) and a transformation \( t : \text{SPEC}_R \Rightarrow \text{SPEC}' \). Now there is a selected transformation \( t' : \text{SPEC} \Rightarrow \text{SPEC}' \) and a selected embedding \( e' : \text{SPEC}_R' \rightarrow \text{SPEC}' \), such that diagram (1) in Fig. 10 becomes an extension diagram. In case of \( t \) also being an embedding, we require the existence of a unique extension diagram (2), called mutual extension diagram.

\[
\begin{array}{c}
\text{SPEC}_R \xrightarrow{e} \text{SPEC} \quad \text{SPEC}_R \xrightarrow{e} \text{SPEC} \\
\downarrow_{(1)} \quad \downarrow_{t'} \\
\text{SPEC}_R' \xrightarrow{e} \text{SPEC}' \quad \text{SPEC}_R' \xrightarrow{e} \text{SPEC}' \\
\end{array}
\]

Fig. 10. Extension
The multiple extension defined below expresses the possibility to apply a set of transformations to a larger context within a single transformation. It differs from the parallel extension used in [16] and [12] by allowing given transformations with the same codomain only, and it contains the extension defined above as a special case. In general, this construction is not available for all families of embeddings and corresponding transformations. Intuitively, such families allow multiple extension, if the boundary of all embeddings is preserved i.e. the transformations do not delete or rewrite parts that are needed to maintain a well-formed specification, and all overlappings with respect to the embeddings are transformed uniquely.

**Definition 3.2 Multiple Extension** Given an index set $I$, a corresponding family of embeddings $e = (e_i : \text{SPEC}_R \rightarrow \text{SPEC})_{i \in I}$ and a family of transformations $tr = (tr_i : \text{SPEC}_R \Rightarrow \text{SPEC}_R)_{i \in I}$. Now $e$ and $tr$ allow multiple extension, if there exist a selected transformation $t : \text{SPEC} \Rightarrow \text{SPEC}'$ and a single embedding $e'_i : \text{SPEC}_R \rightarrow \text{SPEC}'$. We call diagram $(1_i)_{i \in I}$ multiple extension diagram.

$$
\begin{array}{c}
\text{SPEC}_R \\
tr_i \\
(1_i)_{i \in I} \\
t \\
\text{SPEC} \\
\end{array}
\begin{array}{c}
\text{SPEC}_R \\
e_i \\
e \\
\text{SPEC} \\
\end{array}
\begin{array}{c}
\text{SPEC'} \\
b_i \\
\end{array}
$$

Fig. 11. Multiple Extension

**Definition 3.3 Compatibility of Embeddings with Multiple Extension** A family of embeddings $(e_i : \text{SPEC}_R \rightarrow \text{SPEC})_{i \in I}$ is compatible with multiple extension, if for each multiple extension diagram (1) with a family of transformations $(tr_i : \text{SPEC}_R \Rightarrow \text{SPEC}_R)_{i \in C \subseteq I}$, we have for the family of embeddings $(e_j : \text{SPEC}_R \rightarrow \text{SPEC})_{j \in I \setminus C}$ a selected family of embeddings $(e'_j : \text{SPEC}_R \rightarrow \text{SPEC}')_{j \in I \setminus C}$. We require the existence of a subclass $D$ of all families of embeddings such that all elements in this subclass are compatible with multiple extension. Moreover, we require that $D$ is closed under multiple extensions. I.e. if $(e_i : \text{SPEC}_R \rightarrow \text{SPEC})_{i \in I}$ in Fig. 12 is in class $D$ and (1) is an extension diagram and we have embeddings $e'_j$ then also $(e'_j : \text{SPEC}_R \rightarrow \text{SPEC'})_{j \in I \setminus C \cup \{e' : \text{SPEC}_R \rightarrow \text{SPEC}'\}}$ is in class $D$. In the corresponding instantiation this can be achieved by defining $D$ by non-overlapping embeddings. Note that the
instantiation has to define which transformations and embeddings make up a multiple extension diagram.

Moreover, we assume horizontal and vertical composition of multiple extension diagrams: Given diagrams (1), (2), (3) in Fig. 13 with $i \in I$ and $j \in J$. Now (1+3) and (2+3) have to be multiple extension diagrams if (1) and (2) are multiple extension diagrams and (3) is an extension diagram.

3.2 Components and Composition

Based on the requirements explained above, we are now able to define component specifications and the corresponding composition operation.

**Definition 3.4 Component** A component specification $COMP = (BOD, REQ, PRV, req, prv)$ consists of a body specification $BOD$, a family of require specifications $REQ = (REQ_i)_{i \in I}$ for some index set $I$, a family of provide specifications $PRV = (PRV_j)_{j \in J}$ for some index set $J$ and of suiting families of embeddings $req = (req_i : REQ_i \rightarrow BOD)_{i \in I}$ and transformations $prv = (prv_j : PRV \Rightarrow BOD)_{j \in J}$, respectively, where we require that the family of embeddings is in class $D$ and thus compatible with multiple extension in the sense of Def. 3.3.

Note that the components in our example in Sect. 2 fit into this abstract definition. The web browser component in Fig. 3 contains two provide specifications, a body specification and a single require specification. The corresponding transformations of the provide interfaces are shown in Fig. 4.

Next, we summarize the conditions ensuring that a given set of connected components can be reduced to a single component. According to [16,12] we call such a set an architecture. An architecture $A$ is a set of components $COMPS(A) = (COMP_i)_{i \in I}$ and corresponding connecting transformations $CONS(A)$ that fulfill the properties listed below. Each architecture $A$ can be illustrated by an architecture graph $G_A$ (e.g. as for the web browser suite in Fig. 2), obtained by shrinking $A$ to a graph representation that contains nodes labeled by the component names and edges labeled by the connecting transformations.

- There are no isolated components in the architecture.
• Each requirement specification is the source of at most one connecting transformation.

• For each component we require that its embedding of the require interfaces and the connected realizing transformations allow multiple extension.

(i.e. \((\text{prov}_{2y} \circ \text{con}_{1,2,y})_{i \in I, y \in I; C_2, \text{con}_{1,2,y} \in \text{CONS}(A)}\) allow multiple extension)

• There are no cycles in the graph obtained by representing single specifications by nodes and transformations and embeddings by non-directed edges.

In [16,12] it has been shown that architectures can be reduced to a single component, if the applied composition operations yield unique results independent of their application order. This is the case for the operations defined below.

The hierarchical composition with multiple interfaces defined below connects a single providing component to a single requiring component, possibly via different provide and require interfaces. Intuitively, the requiring component is glued with the providing component over the provide interfaces accessed by the requiring component.

**Definition 3.5 Hierarchical Composition with Multiple Interfaces**

Given a requiring component \(\text{COMP}_{i_R} = (BOD_{i_R}, \text{REQ}_{i_R}, \text{PRV}_{i_R}, \text{req}_{i_R}, \text{prov}_{i_R})\) and a providing component \(\text{COMP}_{i_P} = (BOD_{i_P}, \text{REQ}_{i_P}, \text{PRV}_{i_P}, \text{req}_{i_P}, \text{prov}_{i_P})\) with index sets \(I_R, J_R\) and \(I_P, J_P\) for the require and provide interfaces of the requiring component and the providing component, respectively. We denote the index set of the require interfaces actually connected

![Diagram of Hierarchical Composition with Multiple Requirements](image)

Fig. 14. Hierarchical Composition with Multiple Requirements

with the providing component by \(C_{i_R} \subseteq I_{i_R}\). Given corresponding connecting transformations \(\text{con} = \text{con}_{i: \text{REQ}_{i_R}} \Rightarrow \text{PRV}_{i_P(i)}\) with \(i \in C_{i_R} \subseteq I_{i_R}\) and \(i(i) \in J_{i_P}\), such that the family of embeddings \(\text{req}_{i_R}\) and the family of composed transformations \(\text{xcon}_{i} = (\text{prov}_{i_P(i)} \circ \text{con}_{i})_{i \in C_{i_R}}\) allow multiple extension. The mapping \(l: I_{i_R} \rightarrow J_{i_P}\) has to be injective. The index sets of all components are disjoint. In the first step we can derive a multiple extension diagram \((I_{i})_{i \in C_{i_R}}\) with selected transformation \(\text{xcon}'\) and embedding \(\text{req}_{i_R}'\). The compatibility of the embeddings \((\text{req}_{i_R})_{i \in I}\) with respect to multiple extension, which is given by the component definition, yields a set of embeddings
are defined as

\[
(req_{R_z'})_{z \in I_R \setminus C_R}, \text{ such that } (req_{R_z'})_{z \in I_R \setminus C_R} \cup \{req_{R'}\} \text{ is again in class D and thus compatible with respect to multiple extension. Now we define the result of the } \text{Hierarchical Composition with Multiple Interfaces} \text{ (short: composition) by}
\]

\[
COMP_R \circ_{\text{con}} COMP_P = \quad \text{COMP}_G = (BOD_G, REQ_G, PRV_G, req_G, prv_G),
\]

where the index sets of the requirements and provisions of the new component
are defined as 

\[
I_G = (I_R \setminus C_R) \cup I_P \text{ and } J_G = J_R \cup J_P,
\]

respectively. And we have:

\[
REQ_G = (REQ_{R_z'})_{z \in I_R \setminus C_R} \cup \text{REQ}_P
\]

\[
req_G = (req_{R_z'})_{z \in I_R \setminus C_R} \cup (req_{R'} \circ req_{P_y})_{y \in I_R}
\]

\[
PRV_G = PRV_R \cup PRV_P
\]

\[
prv_G = (xcon' \circ prv_{P_{j'}})_{j \in I_R} \cup (\text{trafo}(req_{R'}) \circ prv_{P_x})_{x \in I_P}
\]

Note that the family of embeddings \((req_{G_z})_{z \in I_G}\) is again in class \(D\). In Fig. 15 the elements of the resulting component are depicted in detail.

![Fig. 15. Result of Composition](image)

Since we allow different components accessing the same provide interface, all provide interfaces are preserved by the composition.

Fig. 16 shows the composition of the mailer and the text editor component of our example from Sect. 2. Note that only the transformations \(prv_{TE}, \text{con}_{M, rm2, TE}\) and the embedding \(req_{M_{rm2}}\) are shown in detail.

The hierarchical composition with multiple interfaces is independent of its application order. Since there are three possibilities of overlappings for two composition steps, we present three different theorems: associativity of composition, compatibility of composition I and II.

Whenever we have two connections crossing the same level of a given component architecture, we also offer a parallel composition, which constructs the result of two compositions within a single step. This is the case in the Theorems 3.8 and 3.9.
Theorem 3.6 **Associativity of Composition** Given three components $\text{COMP}_i = (BOD_i, REQ_i, PRV_i, req_i, prv_i)$ for $i \in \{1, 2, 3\}$ as shown in the architecture graph $G_{A1}$ and families of transformations $\text{con}_1 = (\text{con}_{1,i} : \text{REQ}_i \Rightarrow \text{PRV}_{2(l(i))})_{i \in C_1}, \text{con}_2 = (\text{con}_{2,k} : \text{REQ}_i \Rightarrow \text{PRV}_{3(l(k))})_{k \in C_2}$, for some $C_1 \subseteq I_1, C_2 \subseteq I_2$, where $I_1$ and $I_2$ denote the index sets of the require interfaces of $\text{COMP}_1$ and $\text{COMP}_2$, such that the pairs of families $(\text{prv}_{2(l(i))} \circ \text{con}_{1,i})_{i \in C_1}, \text{req}_1$ and $(\text{prv}_{3(l(k))} \circ \text{con}_{2,k})_{k \in C_2}, \text{req}_2$ allow...
multiple extension, each. Then we have the following associativity law:

\[
(COMP_1 \circ_{con_1} COMP_2) \circ_{con_2} COMP_3 = COMP_1 \circ_{con_1} (COMP_2 \circ_{con_2} COMP_3)
\]

**Proof.** Fig. 18 shows the given components and connecting transformations in detail, where we have:

\[
j \in J_1, \quad l_1 : C_1 \rightarrow J_2 \text{ injective},
\]

\[
i \in C_1 \subseteq I_1, \quad k \in C_2 \subseteq I_2,
\]

\[
i' \in I \setminus C_1, \quad k' \in I_2 \setminus C_2, \text{ and}
\]

\[
j' \in J_2 \setminus l(C_1), \quad m \in I_3, x \in J_3.
\]

![Diagram](https://via.placeholder.com/150)

Fig. 18. Associativity of Composition

We are able to construct the multiple extension diagrams (1) and (2) due to the assumptions. The extension diagram (3) exists, because there are no properties required for this construction. The body of the left side of our equation, \((COMP_1 \circ_{con_1} COMP_2) \circ_{con_2} COMP_3\), is constructed by the following steps: First, we have to construct extension diagram (1) to resolve \(con_1\). We know that (2) and (3) are multiple extension diagrams, thus we can construct (2+3) and resolve \(con_2\). For the construction of the body of the left side of our equation, \(COMP_1 \circ_{con_1} (COMP_2 \circ_{con_2} COMP_3)\), the first step is to resolve \(con_2\) using multiple extension diagram (2), and
afterwards resolving con_1 by multiple extension diagram (1+3). Since extension yields unique resulting specifications and transformations, we obtain a unique body $BOD_{123}$.

### 3.3 Compatibility of Composition

In order to ensure a unique reduction of architectures we need to prove that for all kinds of overlappings of components the result of several composition steps is independent of the order of the composition steps. In Thm. 3.6 this was shown for overlappings along an hierarchy. This section deals with composition steps that include overlappings of components of the same hierarchical level. Fig. 19 shows such an architecture. For this case, the result of the given compositions can be constructed within one parallel step. We can prove that the result of this parallel composition is equal to the sequential composition independent of the ordering.

Fig. 19. Architecture Graph $G_{A2}$

#### Definition 3.7 Parallel Composition

Given three components $COMP_i = (BOD_j, REQ_j, PRV_j, req_j, prv_j)$ for $i \in \{1, 2, 3\}$ and families of transformations $con_{-1} = (con_{-1i} : REQ_{-1i} \Rightarrow PRV_{-3(i)})_{i \in \mathcal{C}_{-1}}, con_{-2} = (con_{-2k} : REQ_{-2k} \Rightarrow PRV_{-3(k)})_{k \in \mathcal{C}_{-2}},$ for some $\mathcal{C}_{-1} \subseteq \mathcal{I}_{-1}, \mathcal{C}_{-2} \subseteq \mathcal{I}_{-2},$ such that the pairs $(prv_{-3(i)} \circ con_{-1i})_{i \in \mathcal{C}_{-1}}, req_{-1}$ and $(prv_{-3(k)} \circ con_{-2k})_{k \in \mathcal{C}_{-2}}, req_{-2}$ allow multiple extension, each. Then we construct the multiple extension diagrams (1) and (2) in Fig. 20. Diagram (3) is constructed as mutual extension diagram, including the resulting body $BOD_{123}$. The result of the parallel composition is given by

$$(COMP_{-1}, COMP_{-2}) \circ (con_{-1}, con_{-2}) COMP_{-3} = COMP_{123} = (REQ_{123}, PRV_{123}, req_{123}, prv_{123}),$$

where

$$REQ_{123} = (REQ_{1'})_{i' \in \mathcal{I}_1 \setminus \mathcal{C}_{-1}} \cup (REQ_{2'})_{k' \in \mathcal{I}_2 \setminus \mathcal{C}_{-2}},$$

$$req_{123} = (req_{2''} \circ req_{1'})_{i' \in \mathcal{I}_1 \setminus \mathcal{C}_{-1}} \cup (req_{1''} \circ req_{2'})_{k' \in \mathcal{I}_2 \setminus \mathcal{C}_{-2}},$$

$$PRV_{123} = (PRV_{1j})_{j \in \mathcal{I}_1} \cup (PRV_{2j'})_{j' \in \mathcal{I}_2} \cup (PRV_{3x})_{x \in \mathcal{I}_3},$$

$$prv_{123} = (trafo(req_{2''}) \circ xcon_{-1} \circ prv_{1j})_{j \in \mathcal{I}_1} \cup$$

$$(trafo(req_{1''}) \circ xcon_{-2} \circ prv_{2j'})_{j' \in \mathcal{I}_2} \cup (prv_{3x})_{x \in \mathcal{I}_3}. $$

35
Theorem 3.8 Compatibility of Composition I Given the same components and connecting transformations as in the definition above, then we have the following compatibility law:

\[
\begin{align*}
\text{COMP}_1 \circ_{\text{con}_1} (\text{COMP}_2 \circ_{\text{con}_2} \text{COMP}_3) &= \\
\text{COMP}_2 \circ_{\text{con}_2} (\text{COMP}_1 \circ_{\text{con}_1} \text{COMP}_3) &= \\
(\text{COMP}_1, \text{COMP}_2) \circ_{\text{con}_1, \text{con}_2} \text{COMP}_3 &=
\end{align*}
\]

The web browser suite presented in Sect. 2 involves such a situation: The web browser component and the mailer component access the file browser component, as shown in Figures 2 and 7.

**Proof.** Fig. 20 shows the given setting in detail, where the mappings \( l_1 : I_1 \rightarrow J_3 \) and \( l_2 : I_2 \rightarrow J_3 \) are injective each, but their codomains are not assumed to be disjoint. First, we construct the result of the parallel composition that resolves both connection in a single step. In this case, we start by computing the multiple extension diagrams (1) and (2), which exist due to the assumption of multiple extension for \( \text{prv}_{\beta_1(i)} \circ \text{con}_{\ell_1} \) and \( \text{req}_1 \) as well as for \( \text{prv}_{\beta_2(k)} \circ \text{con}_{\ell_2} \) and \( \text{req}_2 \). Diagram (3) is constructed as mutual extension diagram including the resulting body \( BOD_{123} \). In case of processing only the composition along \( \text{con}_2 \) in the first place, we obtain the extension diagram (2). Afterwards we construct the multiple extension diagram (1’) as depicted in Fig. 21. This diagram is constructed from the same given transformations and embeddings as diagram (1+3) in the case of the parallel composition explained above. This implies \( BOD_{231} = BOD_{123} \).

Analogously we construct \( BOD_{13} \) in the multiple extension diagram (1). Then we construct the extension diagram (2’) which is equal to (2+3) in the parallel composition. This implies \( BOD_{132} = BOD_{123} = BOD_{231} \). Ad-
ditionally, in all three cases we obtain the same families of provide and require interfaces and the corresponding connections, because the disjoint index sets of the given components imply independence of the composition order. \[ \square \]

**Theorem 3.9 Compatibility of Composition II** Given three components \( \text{COMP}_i = (\text{BOD}_i, \text{REQ}_i, \text{PRV}_i, \text{req}_i, \text{prov}_i) \) for \( i \in \{1, 2, 3\} \) and families of transformations \( \text{con}_1 = (\text{con}_{1i} : \text{REQ}_i \implies \text{PRV}_i, \text{req}_i, \text{prov}_i) \), \( \text{con}_2 = (\text{con}_{2k} : \text{REQ}_i \implies \text{PRV}_i, \text{req}_i, \text{prov}_i) \) for \( i \in \{1, 2, 3\} \) and \( k \in C_i \), \( C_1 \subseteq I \) such that the pairs \( (\text{prov}_{2l(i)} \circ \text{con}_{1i}) \subseteq C_i \), \( \text{req}_1 \) and \( (\text{prov}_{2l(k)} \circ \text{con}_{2k}) \subseteq C_i \), \( \text{req}_1 \) allow multiple extension, each. Then we have the following compatibility law:

\[
\begin{align*}
(\text{COMP}_1 \circ \text{con}_1 \text{COMP}_2) \circ \text{con}_2 \text{COMP}_3 &= \text{COMP}_3 \\
(\text{COMP}_1 \circ \text{con}_1 \text{COMP}_3) \circ \text{con}_2 \text{COMP}_2 &= \text{COMP}_2 \\
\text{COMP}_1 \circ (\text{con}_1, \text{con}_2) (\text{COMP}_2, \text{COMP}_3) &= \text{COMP}_3
\end{align*}
\]

In our example in Sect. 2 this situation occurs as well. Figures 2 and 7 show that the mailer component accesses both, the file browser component and the text editor component.

A full proof of the theorem is given in [26]. Here, we only sketch its main idea. In [16] the parallel extension diagram was introduced that embeds independent transformations into a common larger context. Moreover, it offers a special case with all but one given transformation being identities. This is also the case here, because the require interfaces of \( \text{COMP}_1 \) are disjoint. Composing those diagrams yields the intended uniqueness of the body construction.
4 Instantiation to Elementary Nets

In this section we show that the specification technique of elementary nets [35] fits into our generic framework. This includes the definition of embeddings and transformations, and based on that, the construction of the multiple extensions.

The hierarchic transformation-based concept in [13] and the connector component framework in [16] have been instantiated with a variety of specification techniques: HLR-systems and algebraic specifications in [25], Petri nets in [15] and UML diagrams in [12]. Since our approach is a generalization of those two concepts the instantiations can be easily adopted to the new concept.

Elementary transition systems are a special notion of Petri nets, allowing only arcs and place weights of arity one. We use the algebraic notion of Petri nets as given in [29] and extend it by the initial marking. This enables us to use a set based representation of the pre and post functions of the transitions of the nets.

An elementary net $N = (P, T, \text{pre}, \text{post}, m)$ consists of a set of places $P$ and a set of transitions $T$. The functions $\text{pre}, \text{post} : T \to \mathcal{P}(P)$ represent the connecting arcs, and the set $m \subseteq P$ contains all initially marked places. Plain morphisms $f : N_1 \to N_2$ between elementary nets are mappings of places $f_P : P_1 \to P_2$ and of transitions $f_T : T_1 \to T_2$ that are compatible with the $\text{pre}$ and $\text{post}$, i.e. $f_P \circ \text{pre}_1(t) = \text{pre}_2 \circ f_T(t)$ and analogously for $\text{post}$. The mapping has to preserve and reflect the initial marking, i.e. $f_P(m_1) \subseteq m_2$ and $m_2 \setminus f_P(m_1) \subseteq P_2 \setminus f_P(P_1)$. This category $\mathcal{E}N_{\text{plain}}$ has pushouts.

Embeddings are injective morphisms. The following notion of transformation of elementary nets is an adaption of the substitution morphisms of place/transition nets in [30]. These morphisms replace transitions of the original net by whole subnets in the target net and map places injectively. Again, the markings are preserved and reflected. More precisely, a substitution morphism $s = (s_P, s_T) : N_1 \to N_2$ with $N_i = (P_i, T_i, \text{pre}_i, \text{post}_i, m_i)$ for $(i = 1, 2)$ is given by an injective mapping of places $s_P : P_1 \to P_2$ and a mapping $s_T : T_1 \to \mathcal{P}(N_2)$ with $s_T(t) := N_2^t = (P_2^t, T_2^t, \text{pre}_2, \text{post}_2, m_2) \subseteq N_2$ where $\text{pre}_2, \text{post}_2$ and $m_2$ are restricted to the subset of transitions $T_2^t$ and the subset of places $P_2^t$. Again we have preservation and reflection of the marked places, i.e. $s_P(m_1) \subseteq m_2$ and $m_2 \setminus s_P(m_1) \subseteq P_2 \setminus s_P(P_1)$. Composition is well-defined analogously to [30]. So, we have the category $\mathcal{E}N$. Similar to [30] plain morphisms are a special case of substitution morphisms.

We have the extension properties as required in Def. 3.1, because there are in the category $\mathcal{E}N$ pushouts of embeddings with substitution morphisms as well as pushouts of plain morphisms only. The abstract framework requires a class $\mathcal{D}$ with compatibility of embeddings with multiple extension (see Def 3.3). For this instantiation with elementary nets a family of embeddings $e_i : N_i \to N$ has no overlappings, if the codomain of the embeddings
$e_i(N_i)$ are pairwise disjoint.

Then we have multiple extension as required in Def. 3.2. Basically we glue $N \cup R$ with $N$ together by replacing the embeddings of $N \cup R_i$ by their substitution subnets $tr_i(N \cup R_i)$ in $N \cup R$. In the detailed proof [26] we have given the construction in categorical terms, based on the following diagram in $EN$:

$$
\begin{array}{c}
N \cup R_i \xrightarrow{e_i} N \\
\xrightarrow{tr_i} \\
N \cup R \xrightarrow{e_i'} \tilde{N} \xrightarrow{e} N'
\end{array}
$$

First, we construct $i$ pushout diagrams (i). Next we construct the star-pushout $N \cup R_i \xrightarrow{e_i'} \tilde{N} \xrightarrow{e_i} \tilde{N}$ and subsequently the star-coequalizer $N \xrightarrow{\tilde{e} = \tilde{e} \circ \tilde{e}_i \circ \tilde{e}_i'} \tilde{N} \xrightarrow{e} N'$. Then we have the unique $e' = \tilde{e} \circ \tilde{e}_i \circ \tilde{e}_i'$ and the unique $tr = \tilde{e} \circ \tilde{e}_i \circ \tilde{e}_i'$ We show in [26] that this star-coequalizer exists and that $e'$ is a well-defined embedding.

Compatibility of embeddings with multiple extension as required in Def. 3.3 we have for families of embeddings $(e_i : N_i \rightarrow N)_{i \in I}$ that have no overlappings, because the pushout and coequalizer constructions leave those parts that are not in the codomain $(e_i(N \cup R_i))_{i \in C \subseteq I}$ unchanged, especially the codomain of $(e_j(N \cup R_j))_{j \in I \setminus C}$. Hence there is the family of embeddings $(e'_j : N \cup R_j \rightarrow N')_{j \in I \setminus C}$ that remains non-overlapping, see [26].

$$
\begin{array}{c}
N \cup R_i \xrightarrow{e_i} N \\
\xrightarrow{tr_i} \\
N \cup R \xrightarrow{e_i'} \tilde{N} \xrightarrow{e_i} \tilde{N} \xrightarrow{e} N'
\end{array}
$$

5 Conclusion

In this paper we present a generic component concept capable of handling multiple provide and require interfaces and multiple access. This includes the definition of generic components and a hierarchical composition operation for multiple interfaces. Moreover, we introduce the concurrent application of composition steps, called parallel composition. Based on that we prove the result of two overlapping composition steps to be independent of the construction ordering. This induces that the given reduction semantics of architectures is unique. The generic concept is instantiated to the sample specification technique of elementary nets which is also used for the small web browser suite in order clarify the main ideas.

Dynamic software architectures that use formal techniques are investigated in [6,5]. Many of those approaches use graph transformations for the specification of dynamic changes and reconfigurations. In [32] we have integrated the generic component concept with high-level replacement systems, a categorical generalization of graph transformations. This work can be considered
as the technical foundation for the extension of the approach introduced in this paper to dynamic architectures. Since the semantics of architectures is defined by graph transformations we can apply corresponding transformation engines as for example the AGG tool [1] in order to compute the semantics automatically.

As already mentioned in Sect. 3 the handling of multiple accesses in component architectures depends on the used specification technique and the corresponding instance notion. The composition operations presented in this paper are fully adequate for techniques with a loose semantics, i.e. each specification induces a set of valid instances. For techniques with a close semantics there are two possibilities of resolving multiple access. The first variant glues the requiring components over the providing one. This is suitable for a shared access, as used in our example of Sect. 2. Exclusive access requires a different composition operation that creates a copy of the required component for each request. See [26] for details.

References


Extending a Component Specification Language with Time

Björn Metzler and Heike Wehrheim

Institut für Informatik
Universität Paderborn
33098 Paderborn, Germany

Abstract

In a formal approach to component specification, interfaces are usually described using pre- and postconditions of methods or protocols. In this paper we present an approach for integrating time into a component specification language which already allows for pre/post and protocol descriptions. The specification of timing aspects is indispensable when treating components of embedded systems underlying hard real-time requirements. In order to allow for a smooth integration into the existing specification language and to ease reading and writing of interfaces, we do not extend the language with yet another formalism (for time), but instead only add a specific feature (i.e. clocks) to it. We define a semantics for this new specification language in terms of timed automata, which thus also opens the possibility of analysing interface descriptions with the UPPAAL model checker. We furthermore give timed simulation conditions and prove their soundness with respect to inclusion of timed traces, the notion of implementation in timed automata. This implementation relation can be used as a correctness criterion for interoperability and substitutability checks.

Key words: Interface specification, timed automata, pre/post conditions, protocols, simulation, verification.

1 Introduction

Interfaces of components are typically described by giving signature lists, pre- and postconditions of methods or by defining protocols (i.e. valid call sequences). Different approaches and languages have been proposed for these purposes: the signature list only technique is the approach adopted by most industrial middleware platforms, pre- and postconditions are for instance used

1 Email: bmetzler@uni-paderborn.de
2 Email: wehrheim@uni-paderborn.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
in [16,28,18] and protocol definitions for components given as finite state automata, process algebra descriptions or temporal logic can be found in [20,21,11]. For embedded systems, it is however also important to specify timing constraints of interfaces, such as deadlines guaranteed or expected by a component.

In this paper we set out to develop a component specification language which allows for the specification of pre- and postconditions, protocols and timing constraints. The starting point here is an already existing language which contains features for specifying pre- and postconditions and protocols. The notation, called CSP-OZ [9], is a combination of the process algebra CSP [12,22] with the state-based, object-oriented formalism Object-Z [24]. The process algebra is used to specify specific call sequences guaranteed/expected by the component, the state-based formalism handles data-dependent aspects like pre- and postconditions of methods. Both formalisms come with built-in notions of refinement which is the formal development concept guaranteeing substitutability. Thus refinement can be used as correctness criterion for interoperability and substitutability checks. The integrated notation CSP-OZ is now extended to allow for the specification of timing constraints. To this end, we however do not integrate a third formalism into the existing combination but instead only add a specific clock type for Object-Z variables. Clock variables can be declared, queried and changed just like ordinary variables. These clock variables allow for the specification of deadlines, minimum and maximum delays between method calls etc.. This is similar to the way finite automata are extended to timed automata [1], which is the standard formalism for describing systems with timing aspects (they, however, do not allow for a high level description of state-based and behavioural aspects).

For this specification language (called timed CSP-OZ) we furthermore propose a method for analysing component interfaces and we define a formal notion of implementation, which can - like refinement - be used for substitutability checks. The analysis method is based on a semantics for the language in terms of timed automata (or more precisely, timed transition systems, since the semantics will not always yield a finite state automaton). In case of a finite number of states we can then use one of the timed automata model checkers for verification (e.g. Kronos [27] or UPPAAL [3]).

Based on this semantics we can furthermore use the notion of implementation associated with timed automata for timed CSP-OZ. The implementation relation for timed automata is inclusion of timed traces (language inclusion for words with time stamps). We define timed simulation conditions and show their soundness with respect to this relation. This opens the way for a stepwise proof of implementation.

The paper is structured as follows. Next, we start with a simple example of a timed CSP-OZ specification on which we explain the general idea and which will serve as an illustration of the main results in the next sections. Section 3 gives a short introduction to timed automata. We then define the semantics
for timed CSP-OZ specifications in terms of timed automata. In Section 4 we show how to analyse interface specifications in timed CSP-OZ with the timed automata model checker UPPAAL. Section 5 gives timed simulation conditions which can be used to prove language inclusion relationships between interfaces (and thus substitutability). The last section concludes and discusses related work.

2 A first example

The formalism *timed CSP-OZ* that we introduce in this paper is an extension of CSP-OZ with time. CSP-OZ [9] is a combination of the process algebra CSP [12] and the state-based specification formalism Object-Z [24]. It employs CSP to describe aspects of dynamical behavior of components (allowed call sequences of methods); it uses Object-Z to describe data aspects, i.e. the static behavior of operations like pre- and postconditions. For this, Object-Z uses set theory and predicate logic.

We directly give a timed CSP-OZ specification here since these specifications will in general not look very different from plain CSP-OZ specifications. For timed CSP-OZ we always assume a type *Clock* taking values from the set of non-negative reals (the time):

\[ Clock == \mathbb{R}_+ \]

Variables of type clock can be declared as attributes of classes and may (under some restrictions) appear in predicates within method schemas. The following example shows an abstract specification of the interface of a watchdog component. A watchdog component should control a certain method *note* which is to be repeatedly executed within 10 time units after its last occurrence. An alarm can either be raised by a *ring* after the expiration of the deadline or after at least 8 seconds by using a *flash* signal. A component implementing this interface may choose one of these options. Below, the component is specified as an Object-Z class.

```plaintext
Watchdog

method note, ring, flash
main = note → main
   □ (ring → Alarm_r, □ flash → Alarm_f)
Alarm_r = ring → Alarm_r
Alarm_f = flash → Alarm_f
```

3 There are slight differences in the use of Object-Z within CSP-OZ and in the standard definition. In this paper we will, nevertheless, plainly say Object-Z even when meaning the Object-Z part of CSP-OZ.
The class first consists of an enumeration of the set of methods it supplies and uses (usually with their signatures, which are, however, empty in this simple example). Next, a set of CSP process equations (with main process main) gives the protocol of the component. Finally, a number of Z schemas describe the state space, the initialisation and the methods of the class: The watchdog class has three methods note, ring and flash which have neither input nor output parameters. The CSP part specifies that note can be repeatedly executed until either the first ring or the first flash happens. Afterwards, only further ring’s resp. flash’s can follow (□ is the CSP operator for external choice and → the prefix operator describing sequencing). The Object-Z part declares three attributes: alarm for describing that an alarm has been raised and two clocks \( x_r \) and \( x_f \) used for determining whether the timing requirements are met. The class invariant specifies a condition which relates alarm to the clock variable \( x_r \). The three operation schemas define the execution of methods note, ring and flash: the precondition of note is the clock \( x_r \) being less than ten \(^4\), the postcondition specifies both clocks to be zero (\( x_r' \) denotes the value of \( x_r \) in the after state). Method ring can be executed if clock \( x_r \) is greater or equal to 10 and upon execution the alarm is set. The same holds for method flash which may be executed if clock \( x_f \) is greater or equal to 8.

After having introduced timed CSP-OZ by means of a simple interface specification, we are next interested in the analysis of such specifications and in the definition (and the checking) of an implementation relation between specifications. Such a relation could be used for substitutability checks. To

\( ^4 \) Like Object-Z and CSP-OZ, we use a blocking semantics for operations here: the precondition acts as a guard to the method execution.
this end, we will first define a semantics for timed CSP-OZ specifications.

3 Semantics

Protocols of components are quite often described by finite state automata. Here, we choose a similar formalism for our semantics, namely an extension of finite automata to time, called timed automata [1,4]. The timed automaton for a timed CSP-OZ specification will capture the complete behaviour of the specification, including the data dependent aspects covered by Object-Z. Timed automata can, however, not be used to specify pre- and postconditions of methods, thus we do not directly use timed automata for interface specifications, only for their semantics.

3.1 Timed automata

Timed automata are finite automata enhanced with clock variables which can be queried and reset on transitions. To ensure decidability of the emptiness problem (and thus allow for verification), the conditions on clocks are usually restricted. We will later fix similar restrictions on our Z predicates over clocks to ensure that timed CSP-OZ can be safely mapped onto timed automata.

Definition 3.1 Let $X$ be a set of clock variables. The set of clock conditions over $X$, $\Phi(X)$, is given by the following grammar (where $c \in \mathbb{Q}_{\geq 0}$):

$$\varphi ::= x = c \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid (\varphi \land \varphi)$$

We let $\Sigma$ describe the global alphabet of operations of a specification, which we will call events, $\Sigma^*$ the set of finite words, $\Sigma^\omega$ the set of infinite words over $\Sigma$, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and $X$ a global set of clock variables. Since automata always have a finite set of states but our specifications are easily infinite state (due to data) we first define timed (transition) systems as ”timed automata with infinite number of locations” and afterwards have timed automata as a special case of timed systems. A timed system has all the ingredients of a finite state machine (i.e. states or locations, transitions and an initial state), and in addition has a labelling of transitions with clock conditions (determining when the transition can be taken) and sets of clocks (giving all clocks reset upon taking the transition). Furthermore, clock conditions can be associated with locations meaning that the automaton can only be in this location when the clock condition holds.

Definition 3.2 A timed system is a tuple $T = (Q, \rightarrow, q_0, I)$ where

- $Q$ is a (possibly infinite) set of locations,
- $\rightarrow \subseteq Q \times \Sigma \times \Phi(X) \times 2^X \times Q$ are the transitions (or edges),
- $q_0$ is the initial location,
- $I : Q \to \Phi(X)$ assigns invariants to locations.
We write \( q \xrightarrow{a,\varphi,Y} q' \) for \( (q, a, \varphi, Y, q') \in \rightarrow \); the clock condition \( \varphi \) will sometimes also be called the \textit{guard} of the transition, \( a \) its \textit{label} and \( Y \) its \textit{resets}. In case of \( Q \) being a finite set we say that \( T \) is a \textit{timed automaton}. Note that unlike Alur and Dill’s timed automata we have no Büchi acceptance states here, instead progress is achieved by attaching invariants to states. This is a variant of timed automata (used in UPPAAL and Kronos as well) which is sometimes also referred to as timed \textit{safety} automata.

Like finite state automata, timed automata accept languages. In this case, however, languages are sets of timed words (or \textit{timed traces}). In contrast to a language over the alphabet \( \Sigma \) consisting of \( \sigma \in \Sigma^\infty \), a timed language is a set of tuples \( (\sigma, \tau) \in \Sigma^\infty \times \mathbb{R}_+^\infty \):

\[ \text{Definition 3.3} \]

A \textit{timed trace} \( (\sigma, \tau) = (a_1 a_2 \ldots, \tau_1 \tau_2 \ldots) \) is a pair of finite or infinite sequences of events \( a_i \in \Sigma \) and time values \( \tau_i \in \mathbb{R}_+ \) such that \( \tau_{i+1} \geq \tau_i \) for all \( i \geq 1 \). In case of \( \sigma \) and \( \tau \) being finite both sequences furthermore have the same length, i.e. \( \#\sigma = \#\tau \).

An example of a timed trace of our \textit{Watchdog} specification is

\[ \text{note note ring ring ring \ldots, 5.3 14.9 24.9 27.4 33.8 \ldots} \]

The set of timed traces of a timed system can be derived by looking at its possible \textit{configurations}. A \textit{configuration} \( (q, \nu) \) consists of a location \( q \) and a clock valuation \( \nu : X \to \mathbb{R}_+ \). We define two operations on clock valuations which are used to describe the executions of a timed system:

- \textit{time shift}:
  \[ (\nu + d)(x) = \nu(x) + d \]

- \textit{modification}:
  \[ \nu[Y := d](x) = \begin{cases} d, & \text{if } x \in Y, \\ \nu(x), & \text{else.} \end{cases} \]

A timed system is able to perform two kinds of transitions, \textit{delay transitions}, where the time advances while the automaton stays in a location

\[ (q, \nu) \xrightarrow{d} (q, \nu + d) \text{ iff } \nu \models I(q) \land \nu + d \models I(q), \ d \in \mathbb{R}_+ \]

and \textit{action transitions}, where an event-labelled transition is taken

\[ (q, \nu) \xrightarrow{a,\varphi,Y} (q', \nu') \text{ iff } q \xrightarrow{a,\varphi,Y} q' \land \nu \models \varphi, \nu' = \nu[Y := 0], \nu \models I(q), \nu' \models I(q') \]

A timed trace \( (\sigma, \tau) = (a_1 a_2 \ldots, \tau_1 \tau_2 \ldots) \) is in the language \( \mathcal{L}(T) \) of a timed
system $T$ if there is an execution

$$\langle q_0, \nu_0 \rangle \xrightarrow{d_1} \tau_1 \xrightarrow{\nu_1} \langle q_1, \nu_1 \rangle \xrightarrow{d_2} \tau_2 \xrightarrow{\nu_2} \langle q_2, \nu_2 \rangle \rightarrow \ldots$$

where $\tau_i = \tau_{i-1} + d_i, \tau_0 := 0$ and $\nu_0(x) = 0$ for all clock variables $x$. Similar to the untimed case, the language of a timed automaton can be used to define an implementation relation between timed automata. A timed system $T_2$ is said to be an implementation of $T_1$ iff $L(T_2) \subseteq L(T_1)$.

The timed automaton for a timed CSP-OZ specification will be constructed in two steps: First, we derive a timed automaton for the CSP part and the Object-Z part alone. In the second step, these will be combined via parallel composition giving rise to an automaton which obeys the restrictions of both parts. Hence we next define a parallel composition operator $||_A$ on timed automata, describing synchronous parallel composition requiring synchronisation on all actions in the set $A \subseteq \Sigma$:

**Definition 3.4** Let $T_i = (Q_i, \rightarrow_i, q_{0,i}, I_i), i = 1, 2,$ be timed systems over $\Sigma_1$ and $\Sigma_2$ with disjoint set of clock variables $X_1, X_2$, and let $A = \Sigma_1 \cap \Sigma_2$ be the synchronisation set. The synchronous parallel composition of $T_1$ and $T_2$, $T_1 ||_A T_2$, is defined to be the timed system $T = (Q, \rightarrow, q_0, I)$ such that

- $Q = Q_1 \times Q_2$,
- $(q_1, q_2) \xrightarrow{a, \varphi, Y} (q'_1, q'_2)$ iff
  - $a \in \Sigma_1 \cap \Sigma_2$ and
  - $q_i \xrightarrow{a, \varphi_i, Y_i} q'_i, i = 1, 2$, and $\varphi = \varphi_1 \land \varphi_2$, $Y = Y_1 \cup Y_2$ (joint transition), or
  - $a \in (\Sigma_1 \setminus A)$ and $q_1 \xrightarrow{a, \varphi_1, Y_1} q'_1, q'_2 = q_2$, and
  - $a \in (\Sigma_2 \setminus A)$ and $q_2 \xrightarrow{a, \varphi_2, Y_2} q'_2, q'_1 = q_1$,
- $q_0 = (q_{0,1}, q_{0,2})$,
- $I(q_1, q_2) = I_1(q_1) \land I_2(q_2)$.

This now allows us to give a semantics to a timed extension of CSP-OZ.

### 3.2 Semantics of timed CSP-OZ

First, we have to precisely specify what kinds of predicates over clock variables are allowed in specifications. As already mentioned, we assume to have a type $\text{Clock}$ given with values from the nonnegative reals $\mathbb{R}_+$. Every timed CSP-OZ specification may contain a number of clock variables $x_1, \ldots, x_n$ and a number of variables $v_1, \ldots, v_m$ of other types. We then impose the following restrictions on the use of clock variables:

- the init schema specifies all clocks to be initially zero (and furthermore to uniquely fix values for the other variables, in order to have a unique initial state$^5$);

$^5$ This restriction can easily be lifted.
operation schemas may contain clock conditions over unprimed clock variables plus predicates of the form \( x'_i = 0 \) (since in terms of modifications of timed automata, clocks may only be queried and reset);

we need to ensure that the state schema must not have predicates over clock variables other than of the following form:

\[
p \Rightarrow \varphi
\]

where \( \varphi \) is a clock condition and \( p \) a predicate over variables \( v_1, \ldots, v_m \). This restriction is necessary for a unique assignment of invariants to locations of the timed automata.

These conditions ensure that timed CSP-OZ specifications can be mapped onto timed automata. For defining the semantics we next have to separate the clockless parts of the specification from those with clocks. We define \( cl(schema) \) to be the clockless part of a schema, i.e. the declarations and predicates over non-clock variables, \( cc(schema) \) to be the clock part of a method schema, \( cinv(schema) \) to be the predicate(s) of the state schema relating clock variables and other variables, and \( reset(schema) \) to be the set of clocks \( x_i \) with predicates \( x'_i = 0 \) in the schema. The events (or actions) of the timed automaton of a CSP-OZ specification will always have the form \( Op.i.o \) where \( Op \) is the name of a method and \( i \) and \( o \) (possibly omitted) are values for input and output parameters. This is the CSP view on events. Among others, in our example the separation of the Object-Z-part leads to the following schemas (note that \( reset(note) = \{ x_r, x_f \} \) and \( reset(ring) = \emptyset \)):

\[
\begin{array}{c}
cl(ring) \\
\Delta(alarm) \\
alarm' \\
\end{array}
\quad
\begin{array}{c}
cc(note) \\
\Delta(x_r, x_f) \\
x_r < 10 \\
x'_r = 0 \land x'_f = 0 \\
\end{array}
\quad
\begin{array}{c}
cinv(state) \\
\neg alarm \Rightarrow x_r \leq 10 \\
\end{array}
\]

where \( state \) identifies the state schema of the class \( Watchdog \).

The locations of the timed system are (partly) the set of valuations (bindings) of nonclock variables. For such a valuation \( q \) and a state schema \( st \) we write \( q \models st \) if the valuation satisfies the predicates in \( st \); for valuations \( q, q' \), input value \( i \), output value \( o \) and operations schema \( Op \) we write \((q, i, o, q') \models Op \) if \( q \) as before and \( q' \) as after state together with input \( i \) and output \( o \) satisfy the predicate in \( Op \). For understanding the semantics of a timed CSP-OZ class (and for defining timed simulations later) it is useful to think of every class as implicitly having an (infinite) number of operations
Delayₜ for every \( d \in \mathbb{R}_+ \) (a nonnegative real)

\[
\text{Delay} \triangleq [\Delta(x_1, \ldots, x_n) \mid x'_i = x_i + d]
\]

(\( x_1, \ldots, x_n \) the set of all clock variables of the class), advancing the time for \( d \) time units.

The semantics for timed CSP-OZ is now derived in the already mentioned two steps: first, we separately derive a semantics for the part without CSP process equations (called timed Object-Z) and for the CSP part, and in a second step these are combined using the above defined parallel composition operation on timed systems.

**Definition 3.5** Let \( OZ = (State, Init, (Op_i)_{i \in I}) \) be a timed Object-Z class with clock variables \( X = \{x_1, \ldots, x_n\} \) and ordinary variables \( \text{Var} = \{v_1, \ldots, v_m\} \).

The semantics of \( OZ \), \([OZ]\), is the timed system \( T = (Q, \rightarrow, q_0, I) \) with

- \( Q = \{ \rho : \text{Var} \to D \mid \rho |= \text{cl}(State), \rho \text{ type correct} \} \) (\( D \) a domain for values),
- \( q \xrightarrow{Op_i,o,\varphi,Y} q' \) iff \( (q, i, o, q') |= \text{cl}(Op), \varphi = \text{cc}(Op) \) and \( Y = \text{reset}(Op) \),
- \( q_0 |= \text{Init} \),
- \( I(q) = \bigwedge_{\varphi \in \Phi(X)} \exists p : p \Rightarrow \rho \in \text{cmv}(State) \land q = p \} \varphi \)

Figure 1 shows the timed automaton for the Object-Z part of class Watchdog.

Note that the semantics generates a particular class of timed automata: all transitions labelled with the same event have the same guards and clock conditions. Because of simplicity, we have restricted the clock conditions of a method to be state independent. However, an extension of the semantics to clock conditions relating clock variables and predicates over non-clock variables can be achieved.

The definition of the semantics gives us a close correspondence between the states of the timed Object-Z specification and the configurations of the timed automaton which separate clock valuations from clockless valuations. Given a configuration \( \langle q, \nu \rangle, q : \text{Var} \to D, \nu : X \to \mathbb{R}_+ \), we let \( q \oplus \nu : \text{Var} \cup X \to D \cup \mathbb{R}_+ \) denote the valuation combining the two separate valuations. Then we
get the following relationship between a timed Object-Z specification and its
timed automaton:

**Proposition 3.6** Let $OZ = (\text{State}, \text{Init}, (Op_i)_{i \in I})$ be a timed Object-Z class
and $T = (Q, \to, q_0, I) = [OZ]$ its semantics. Then the following holds:

(i) $q_0 \oplus \nu_0 \models \text{Init}$,

(ii) $\forall q, \nu, d : (\langle q, \nu \rangle \xrightarrow{d} \langle q, \nu + d \rangle) \iff (q \oplus \nu, q \oplus (\nu + d)) \models \text{Delay}_d$

(iii) $\forall q, q', \nu, \nu', Op, i, o : (\langle q, \nu \rangle \xrightarrow{\text{Op},i,o} \langle q', \nu' \rangle) \iff (q \oplus \nu, i, o, q' \oplus \nu') \models \text{Op}$

**Proof:**

(i) $q_0 \oplus \nu_0 \models \text{Init}$ follows by the definition of $q_0$ and the fact that we have
restricted initialisation of clock variables to 0.

(ii) Implication from left to right:

$\langle q, \nu \rangle \xrightarrow{d} \langle q, \nu + d \rangle$

$\Rightarrow \nu \models I(q), \nu + d \models I(q), q \models \text{cl(State)}$ (by definition of the semantics)

$\Rightarrow \nu \models \bigwedge_{\varphi : p = \varphi \in \text{cine(State)}, q = p} \varphi$

$\wedge \nu + d \models \bigwedge_{\varphi : p = \varphi \in \text{cine(State)}, q = p} \varphi$

$\Rightarrow q \oplus \nu \models \text{State} \wedge q \oplus (\nu + d) \models \text{State}$

$\Rightarrow (q \oplus \nu, q \oplus (\nu + d)) \models \text{Delay}_d$.

Reverse direction:

$(q \oplus \nu, q \oplus (\nu + d)) \models \text{Delay}_d$

$\Rightarrow \nu \models I(q), \nu + d \models I(q)$

$\Rightarrow \langle q, \nu \rangle \xrightarrow{d} \langle q, \nu + d \rangle$.

(iii) Implication from left to right:

$\langle q, \nu \rangle \xrightarrow{\text{Op},i,o} \langle q', \nu' \rangle$

$\Rightarrow \exists \varphi, Y : q \xrightarrow{\text{Op},i,o,\varphi,Y} q', \nu \models I(q'), \nu \models \varphi, \nu' = \nu[Y := 0], \nu' \models I(q')$

$\Rightarrow (q, i, o, q') \models \text{cl(Op)}, \varphi = \text{cc(Op)}, Y = \text{reset(Op)}, \nu \models \varphi,$

$\nu' = \nu[Y := 0], \nu' \models I(q), \nu \models I(q)$

$\Rightarrow (q \oplus \nu, i, o, q' \oplus \nu') \models \text{Op}$.

Reverse direction:

$(q \oplus \nu, i, o, q' \oplus \nu') \models \text{Op}$ and $\varphi = \text{cc(Op)}, Y = \text{reset(Op)}$

$\Rightarrow \nu' = \nu[Y := 0], \nu \models \varphi, \nu \models I(q), \nu' \models I(q'), q \xrightarrow{\text{Op},i,o,\varphi,Y} q'$

$\Rightarrow \langle q, \nu \rangle \xrightarrow{\text{Op},i,o} \langle q', \nu' \rangle$.

The timed system for the CSP part is very simple (no clock conditions at all) and can be derived using the operational semantics of CSP (referred to as $\to_{\text{CSP}}$ in the next definition) [22]. Note that we do not consider internal
events here, thus we restrict ourselves to deterministic CSP processes. This
choice is influenced by our notion of implementation which is trace refinement.
A distinction between external and internal choice would only be reasonable
in the context of a more discriminable semantic model such as the failures-
divergences model of CSP, which would thus however have to be extended to
the timed setting.

**Definition 3.7** Let \( \text{main} \) be the main process of the CSP part of a timed CSP-OZ class. The *semantics of \( \text{main} \), \([\text{main}]\)*, is the timed system \( T = (Q, \rightarrow, q_0, I) \) over an empty set of clock variables where

- \( Q \) is the set of CSP terms,
- \( q \xrightarrow{\text{Op}.i.o,\text{true},\emptyset} q' \) iff \( q \xrightarrow{\text{Op}.i.o}^{\text{CSP}} q' \),
- \( q_0 = \text{main} \) and
- \( I(q) = \text{true} \) for all \( q \in Q \).

Figure 2 shows the timed automaton for the CSP part of class *Watchdog*.

The semantics of a timed CSP-OZ specification \( C \) consisting of CSP part \( \text{main} \) and Object-Z part \( \text{OZ} \) is then obtained by combining the semantics of the separate parts using the CSP parallel composition operator on timed systems: \([C] = [\text{OZ}] \parallel_A [\text{main}]\). The synchronisation set \( A \) is the intersection of the alphabets of the CSP part and the Object-Z part. The full timed automaton describing the semantics of class *Watchdog* is given in Figure 3.

The class invariant is now a location invariant (for location \( \neg \text{alarm} \)), the preconditions of operations referring to clocks are clock conditions on transitions, the predicates \( x'_r = 0 \) and \( x'_f = 0 \) become clock resets, and CSP part as well as the clockless part of Object-Z determine the structure of the automaton and the labelling of transitions.

### 4 Verification

The formal semantics gives us the possibility of analysing interface specifications. Provided the timed automaton has a finite number of locations (which is the case in our example) we can even use a model checker for the analysis.
Several model checkers for timed automata exist; here, we will use UPPAAL [3]. To do so, we first have to describe the timed automaton in the format required by UPPAAL, and then have to formulate (and check) the properties we are interested in.

For the first part, we have to make some global declarations of channels and clocks for UPPAAL. Thus, for our watchdog we declare three channels \texttt{note}, \texttt{ring} and \texttt{flash} corresponding to the methods of the CSP-OZ class. We also define two global clocks \(x_r\) and \(x_f\) representing the clocks \(x_r\) and \(x_f\) in our specification.

Next and according to our timed automata of Figure 3, we add a template \texttt{Watchdog} (a skeleton of a timed automaton) to our UPPAAL system. Here, we renamed its states (the names of locations have no influence on the set of timed traces) to facilitate formulation of our verification properties. Each transition is labelled with its corresponding channel name\(^6\). Finally we add invariants to the states and clock conditions and resets to the transitions as depicted in Figure 3. Figure 4 shows a screenshot of the automaton representing the timed system of our CSP-OZ specification \texttt{Watchdog}.

This now allows us to automatically analyse our interface specification, for instance check whether our three views on the interface (pre/postconditions, protocols and time) in combination give us the desired behaviour. Here, we prove three requirements on our system. The first is deadlock freedom (which

\(^6\) For simulation and verification purposes, we then later add a second tester component. This component only contains one state and three transitions which synchronize with the three respective channels, i.e. it has no own behavior. This is required since UPPAALs CCS-like synchronisation [19] always expects a partner for a transition labelled with a channel name.
means that it is not possible to find a trace for our timed automaton so that no progress is possible). Deadlock freedom is essential in two different aspects: Considering time constraints, the state invariants and time guards must guarantee that progress is always possible. For example, if we change the invariant on the first ring-transition to \( x_r \geq 11 \), deadlock freedom would not be ensured: In the open time interval \([10, 11]\), neither any transition is possible nor is it possible to stay in location nalarm\(_C1\). The second aspect – which we do not consider here – is communication between more than one component, where deadlocks based on the CSP part of the classes may occur.

Since the query language of UPPAAL is a subset of CTL [6] (i.e. UPPAAL allows for more than proofs of language inclusion), we show that the formula

\[ A \left[ \right] \text{not deadlock} \]

holds. By describing the desired behavior of Watchdog, we want to guarantee that after a certain time, an alarm is raised. That means that we must leave any state representing alarm = false after at most 10 time units. Therefore we have to verify the formula

\[ A[\] x_r>10 \text{ imply not (Watchdog.nalarmM or Watchdog.nalarmC1 or Watchdog.nalarmC2)} \]

which holds as well. Finally, we do not want the alarm to be raised before 8 time units have passed. We verify that

\[ A[\] (Watchdog.alarmR or Watchdog.alarmF) imply x_f>=8 \]

holds.
Next, we are interested in showing language inclusion, i.e. an implementation relationship between a higher level timed CSP-OZ specification $A$ and its implementation $C$. Language inclusion is the correctness criterion that we intend to use for substitutability and interoperability checks of component interfaces. For instance, if we are given two component specifications $A$ and $C$ defining what the components require from other components, $A$ can be safely replaced by $C$ if the language of $C$ is a subset of that of $A$ (it requires less).

Language inclusion for timed automata is in general undecidable [1]. Here, we give an approach to checking language inclusion of timed CSP-OZ specifications. The additional structure present in the specifications allow for a compositional language inclusion check, separately treating the CSP and the Object-Z part. The check for the Object-Z part has to be carried out manually, the check on the CSP part can be done by a CSP model checker.

We start with developing timed simulation conditions for timed Object-Z specifications. These can be used to carry out an operation-wise proof of language inclusion between two timed Object-Z specifications. The conditions can be seen as one half of timed bisimulations [26], and are similar to those in approaches specifying components by pre- and postconditions of their operations. For showing a simulation relation between an abstract and a concrete Object-Z specification we have to give a relation $R$ relating the state spaces of both specifications:

**Definition 5.1**

Let $A = (AInit, AState, (AOp_i)_{i \in I})$ and $C = (CInit, CState, (COp_i)_{i \in I})$ be two timed Object-Z specifications. $C$ is a timed simulation of $A$ if there is a relation $R : CState \leftrightarrow AState$ such that the following hold:

(I) $\forall CState \bullet CInit \Rightarrow \exists AState \bullet AInit \land R$

(C1) $\forall CState, CState', AState \bullet R \land COp \Rightarrow \exists AState' \bullet AOp \land R'$

(C2) $\forall CState, CState', AState, d \bullet R \land Delay^C_d \Rightarrow \exists AState' \bullet Delay^A_d \land R'$

Condition (I) is an initialisation condition requiring every initial state of $C$ to have a corresponding initial state in $A$. Conditions (C1) and (C2) are correctness conditions which state that both delay and action steps of $C$ have corresponding steps in $A$. Note that unlike downward or backward simulation conditions for data refinement we do not have separate applicability conditions here. This is justified by the notion of implementation for timed automata which is simply language inclusion. Language inclusion does not require availability of methods to carry over from the abstract to the concrete system. Thus correctness is sufficient here. Note that the rules are sound but not complete for language inclusion.
Theorem 5.2
Let \( A = (A_{\text{Init}}, A_{\text{State}}, (A_{\text{Op}_i})_{i \in I}) \) and \( C = (C_{\text{Init}}, C_{\text{State}}, (C_{\text{Op}_i})_{i \in I}) \) be two timed Object-Z specifications. Then, the following holds:

If \( C \) is a timed simulation of \( A \) then \( \mathcal{L}(\{C\}) \subseteq \mathcal{L}(\{A\}) \).

**Proof:** The proof relies on Proposition 3.6 showing the correspondence between the states of a timed Object-Z specification and the configurations of its timed automaton.

Assume \((\sigma, \tau) \in \mathcal{L}(\{C\})\). Then there is an execution of \([C]\) over \((\sigma, \tau)\), i.e.

\[
\langle q_0^C, \nu_0^C \rangle \xrightarrow{d_1} \langle q_1^C, \nu_1^C \rangle \xrightarrow{a_1} \langle q_2^C, \nu_2^C \rangle \ldots
\]

such that \(\tau_i = \tau_{i-1} + d_i\) for all \(i > 1\). We inductively construct an execution of \([A]\) over \((\sigma, \tau)\).

Induction base: By Proposition 3.6 we get \( (q_0^C \oplus \nu_0^C) \models C_{\text{Init}} \). Hence, by (I) there is a state \(q_0^A\) such that \( (q_0^A \oplus \nu_0^A) \models A_{\text{Init}} \) and \( (q_0^C \oplus \nu_0^C, q_0^A \oplus \nu_0^A) \in R \).

Induction hypothesis: We assume to have constructed an execution sequence of \( A \) up to some index \(i\)

\[
\langle q_0^A, \nu_0^A \rangle \xrightarrow{d_1} \langle q_1^A, \nu_1^A \rangle \xrightarrow{a_1} \langle q_2^A, \nu_2^A \rangle \ldots \langle q_i^A, \nu_i^A \rangle
\]

such that \(\forall j \leq i : (q_j^C \oplus \nu_j^C, q_j^A \oplus \nu_j^A) \in R \).

Induction step: Assume \((q_i^C \oplus \nu_i^C, q_i^A \oplus \nu_i^A) \in R \). If \( (q_i^A, \nu_i^A) \xrightarrow{a_i} (q_i^C, \nu_i^C + d) \) then by Proposition 3.6 \((q_i^C \oplus \nu_i^C, q_i^C \oplus (\nu_i^C + d)) \models \text{Delay}_d^C \). By (C2) it follows that there is some state \(s^A\) such that \((q_i^A \oplus \nu_i^A, s^A) \models \text{Delay}_d^A \) and \((q_i^C \oplus (\nu_i^C + d), s^A) \in R \). By definition of the Delay operation it follows that \(s^A = q_i^A \oplus (\nu_i^A + d)\). Again by Proposition 3.6 we thus get \( (q_i^A, \nu_i^A) \xrightarrow{a_i} (q_i^A, \nu_i^A + d) \). The case of action transitions can be shown in an analogue way. The fact that this is an execution of \( A \) over \((\sigma, \tau)\) follows since the same \(d_i\) and \(a_i\) appear in the execution of \( A \) (and hence the \(\tau_i\) will be the same). \(\Box\)

We will exemplify timed simulations by the following implementation \textit{ImpWatchdog} of class \textit{Watchdog} which resolves some of the nondeterminism and uses one clock only. The timed language of \textit{ImpWatchdog} is a subset of that of \textit{Watchdog}.

```
<table>
<thead>
<tr>
<th>ImpWatchdog</th>
</tr>
</thead>
<tbody>
<tr>
<td>method flash</td>
</tr>
<tr>
<td>method note</td>
</tr>
<tr>
<td>main = note -&gt; main □ flash -&gt; Alarm</td>
</tr>
<tr>
<td>Alarm = flash -&gt; Alarm</td>
</tr>
</tbody>
</table>
```

57
A timed simulation between ImpWatchdog and Watchdog can be shown with the following representation relation between states of ImpWatchdog and Watchdog:

\[
\begin{array}{c|c}
\text{WatchdogState} & \text{ImpWatchdogState} \\
\text{alarm} & x_f = x \\
& x_r = x \\
\end{array}
\]

The proof involves a number of steps:

1. **Initialisation** amounts to proving

   \[
   \forall \text{warning}, x \cdot \neg \text{warning} \Rightarrow \exists \text{alarm}, x_r, x_f \cdot \neg \text{alarm} \land x_r = 0 \land x_f = 0 \land R
   \]

   which holds (use \(\neg \text{alarm} \land x_r = 0 \land x_f = 0\)).

2. **Correctness for operation note** is

   \[
   \forall \text{warning}, x, \text{warning}', x', \text{alarm}, x_r, x_f \cdot \neg \text{warning} \land x < 8 \land x' = 0
   \Rightarrow \exists \text{alarm}', x_r', x_f' \cdot \neg \text{alarm} \land x_r < 10 \land x_r' = 0 \land x_f' = 0 \land R'
   \]

   which holds by using \(\neg \text{alarm}' \land x_r' = 0 \land x_f' = 0\).

3. **Correctness for operation flash** is

   \[
   \forall \text{warning}, x, \text{warning}', x', \text{alarm}, x_r, x_f \cdot R \land x \geq 8 \land \text{warning}'
   \Rightarrow \exists \text{alarm}', x_r', x_f' \cdot x_f \geq 8 \land \text{alarm}' \land R'
   \]

   where \(\text{alarm}' \land x_f' = x' \land x_r' = x'\) is a solution.

4. **Although there are an infinite number of operations Delay_d** (one for...
each \( d \), there is just one proof to be done, namely

\[
\forall \text{warning}, x, \text{warning}' , x', \text{alarm}, x_r, x_f, d \bullet \\
R \land \text{warning}' = \text{warning} \land x' = x + d \\
\Rightarrow \exists \text{alarm}', x_r', x_f' \bullet \text{alarm}' = \text{alarm} \land x_r' = x_r + d \land x_f' = x_f + d \land R'
\]

which holds immediate, since clock values are all equal. Thus class \text{ImpWatchdog} is indeed an implementation of class \text{Watchdog}.

This so far only gives us a means for showing an implementation relationship between pure timed Object-Z specifications. In addition, we have to look at the CSP parts and have to check whether an implementation relationship between the timed automata of the processes \text{main}_A and \text{main}_C (where \text{main}_A is the CSP process of the abstract class \text{A} and \text{main}_C that of the more concrete class \text{C}) holds as well. Here, the check is very easy: it amounts to checking \text{trace refinement} between the CSP processes. Trace refinement is one of the notions of refinement supported by the semantic model of CSP, and – given the CSP processes are finite state – can be checked using the CSP model checker FDR \cite{10}. Trace refinement is just language inclusion (again assuming no acceptance states) in the timeless setting. Since the timed automata for the CSP processes pose no restrictions on time, trace refinement is sufficient:

\textbf{Lemma 5.3} Let \text{main}_A and \text{main}_C be the CSP processes of timed CSP-OZ classes \text{A} and \text{C}. If \text{main}_C is a trace refinement of \text{main}_A then \( \mathcal{L}(\text{main}_C) \subseteq \mathcal{L}(\text{main}_A) \).

Finally, these two results have to be integrated into one. So far, we have some means for showing an implementation relationship for the timed Object-Z parts and for the CSP parts. These techniques can separately be applied to specifications if the implementation relationship is preserved under the operators that we use for combining the semantics of the separate parts, namely under parallel composition. The following theorem states exactly this property.

\textbf{Theorem 5.4} Let \( S_i = (P_i, \rightarrow_i^S, p_{0,i}, I_i) \), \( i = 1, 2 \), be timed systems over \( \Sigma_i \) and \( T_i = (Q_i, \rightarrow_i^T, q_{0,i}, J_i) \), \( i = 1, 2 \), over \( \Sigma_i \) with a disjoint set of clock variables \( X_i \) for \( S_i \) and \( Z_i \) for \( T_i \). Let \( A = \Sigma_1 \cap \Sigma_2 \) be the synchronisation set. Then the following holds:

\[
\mathcal{L}(S_1) \subseteq \mathcal{L}(T_1) \iff \mathcal{L}(T_2) \subseteq \mathcal{L}(T_1 \upharpoonright_A S_1) \subseteq \mathcal{L}(T_2 \upharpoonright_A S_2)
\]

The proof of this theorem relies on a certain property of delay transitions (that itself depends on the form of clock conditions for invariants) which allows us to combine and decompose delay transitions. Here, we just state and give the proof of this property.

\textbf{Lemma 5.5} Let \( T = (Q, \rightarrow, q_0, I) \) be a timed system. Then

\[ \ldots \]
∀ d, d_1, d_2 \cdot d = d_1 + d_2 \cdot
greaterdot{q, \nu} d_1 \leftrightarrow \langle q, \nu + d_1 \rangle \leftrightarrow \langle q, \nu + d_2 \rangle \leftrightarrow \langle q, \nu + d_1 + d_2 \rangle

**Proof:** The direction $\Leftarrow$ follows immediately since by definition one can assume that the intermediate configuration $\langle q, \nu + d_1 \rangle$ can be skipped.

Let $\langle q, \nu \rangle \rightarrow (q, \nu + d_1)$ and $d = d_1 + d_2$ with $d_1, d_2 \in \mathbb{R}_+$. By assumption we know that $\nu \models I(q)$ and $\nu + d_1 + d_2 \models I(q)$. We need to prove that

$\langle q, \nu \rangle \rightarrow (q, \nu + d_1) \rightarrow (q, \nu + d_2) \rightarrow (q, \nu + d_1 + d_2),$

meaning $\nu + d_1 \models I(q)$. We do this by induction on the $\varphi \in \Phi(X)$:

- **Induction base:**
  - $\varphi = (x = c)$:
    - By $\nu \models x = c$ and $\nu + d_1 + d_2 \models x = c$ we get $d_1 = d_2 = 0$, i.e. $\nu + d_1 \models x = c$, since we do not reset the time in between our delays.
  - $\varphi = (x \leq c)$:
    - We have $\nu \models x \leq c$ and $\nu + d_1 + d_2 \models x \leq c$. Again we immediately deduce $\nu + d_1 \models x \leq c$
  - $\varphi = (x \geq c)$, $\varphi = (x < c)$, $\varphi = (x > c)$:
    - The same arguments are used to show these properties.

- **Induction step:** Let $\varphi = \psi_1 \land \psi_2$. Since $\nu \models \varphi$ and $\nu + d_1 + d_2 \models \varphi$, it follows that $\nu \models \psi_1$, $\nu \models \psi_2$, $\nu + d_1 + d_2 \models \psi_1$ and $\nu + d_1 + d_2 \models \psi_2$. By induction hypothesis we deduce that $\nu + d_1 \models \psi_1$ and $\nu + d_1 \models \psi_2$, i.e. $\nu + d_1 \models \psi_1 \land \psi_2$.

□

Finally, we can combine Theorem 5.2 and Lemma 5.3 in the following way:

**Corollary 5.6** Let $\text{main}_A$ and $\text{main}_C$ be the CSP processes of timed CSP-OZ classes $A$ and $C$, let $\text{OZ}_A$ and $\text{OZ}_C$ be their timed Object-Z-parts. If $\text{main}_C$ is a trace refinement of $\text{main}_A$ and $\text{OZ}_C$ is a timed simulation of $\text{OZ}_A$, then $L([C]) \subseteq L([A])$, i.e. $C$ is an implementation of $A$.

We thus can separately check for language inclusion.

6 Conclusion

In this paper we have proposed an extension of CSP-OZ with features for specifying timing constraints on components. The extension has been minimal in the sense that we neither added a third formalism to CSP-OZ nor exchanged one of the notations which have already been integrated into CSP-OZ. The only extension is a new type for variables in Object-Z. We believe this to be an extension which makes reading and writing of specification particularly easy: once a designer is familiar with CSP and Object-Z he/she can easily read and write timed CSP-OZ specifications.
For this new formalism we gave a formal semantics, showed, by means of an example, how existing model checkers can in principle be used to analyse components interfaces and discussed simulation conditions for implementation relationships. This now gives us a high-level specification language for components which offers a richer set of facilities for modelling than timed automata do. The additional structure present in timed CSP-OZ specifications could furthermore facilitate static analysis of specifications which might prove fruitful for verification. By restricting our semantics to the traces model, we obtain a simple definition of timed simulations which is very close to the techniques and definitions used in the context of timed automata. Nevertheless we want to deal with parallel composition of several components which calls for a more precise semantics. Future work sees the expansion of our approach to the failures-divergences model of CSP [23] and the use of parallel composition and nondeterminism.

Related work.

The combination of an already existing formalism with time is subject of intense research and most often used in the context of safety-critical component-based systems. For example, [15] use OCL for specifying contracts with time and also use an underlying semantics over timed automata.

There are a number of existing integrated notations like CSP-OZ which allow for the description of timing requirements. The approach most often chosen is that of a combination with timed CSP [23]: [17] and [7] combine Object-Z with timed CSP and [25] combines Z with timed CSP. The main difference to these combinations is our semantic model of timed automata. This immediately allows us to use existing standard model checkers for timed systems (when our specifications are finite state).

The use of timed automata (in combination with timed MSCs) for describing components has also been proposed in [5]. Timed automata are therein used for describing individual components and timed MSCs the interaction between components. The two formalisms are, however, not integrated so far; there is no semantics describing the meaning of this combined use of timed automata and MSCs. An approach which uses timed automata like structures to define timed connectors of components can be found in [2].

The two approaches most closest to us are CSP-OZ-DC [14] and a recent combination of Object-Z with timed automata [8]. The latter is an extension of CSP-OZ with Duration Calculus, which is an interval logic for describing timing requirements. The main difference to our work can be seen in the style of specification: while Duration Calculus allows for a declarative formulation of timing constraints, our approach takes a more operational style. The semantics of CSP-OZ-DC is formulated in terms of phase-event-automata in which clocks then explicitly appear. Model checking on this type of automata can be carried out using a constraint-solving based checker [13]. Since our timed automata are close to phase-event-automata a similar type of verification could
be possible for timed CSP-OZ.

The combination of Object-Z and timed automata proposed in [8] also adds clock variables to Object-Z. These variables may however only be used in the timed automaton which in addition may appear in an Object-Z class. Refinement or implementation relationships are so far not treated.

References


Parametric Performance Contracts: 
Non-Markovian Loop Modelling 
and an Experimental Evaluation

Heiko Koziolek¹ Viktoria Firus²

Graduate School Trustsoft 
University of Oldenburg 
26121 Oldenburg, Germany³

Abstract

Even with today's hardware improvements, performance problems are still common in many software systems. An approach to tackle this problem for component-based software architectures is to predict the performance during early development stages by combining performance specifications of prefabricated components. Many existing methods in the area of component-based performance prediction neglect several influence factors on the performance of a component. In this paper, we present a method to calculate the performance of component services while including influences of external services and different usages. We use stochastic regular expressions with non-Markovian loop iterations to model the abstract control flow of a software component and probability mass functions to specify the time consumption of internal and external services in a fine grain way. An experimental evaluation is reported comparing results of the approach with measurements on a component-based webservice. The evaluation yields that using measured data as inputs, our approach can predict the mean response time of a service with less than 2 percent deviation from measurements taken when executing the service in our scenarios.

Key words: performance prediction, parametric performance contracts, service time distribution, software components, stochastic regular expressions, non-Markovian loops

1 Introduction

Despite the rapidly growing performance (i.e. time efficiency in terms of response time and throughput) of computer hardware, performance problems can still be observed in many large hardware/software systems today. One reason for these
Persisting problems is the ever-growing complexity of software. Large software systems have to deal with complex user requirements and are often designed with architectures that do not scale well, even with additional hardware.

Component-based software systems [1] might offer major improvements for engineering software. Lots of research is directed at analysing the performance of component-based software architectures during early development stages. The aim is to predict the performance of a whole architecture by combining the performance specifications of single prefabricated components. Component developers shall provide these specifications in a parameterisable form, so that the software architects or component assemblers can feed them into tools to gain an estimation on the expected performance of their design. This way, design decisions regarding components shall be supported and component assemblers shall be enabled to compare functional equivalent components for their non-functional properties.

To specify the performance of a component, multiple external influences have to be considered, because components shall be third-party deployable [1]. Components may execute external services to provide their services, they can be deployed on different hardware, operating systems and middleware platforms. Furthermore, they interact with their environment by using limited resources of a system, and they are possibly executed by very different user groups. Any approach aiming at predicting performance properties of component-based systems has to take all of these factors into account to be precise. Most of the existing approaches fall short in at least one of the mentioned factors [2].

In this paper, we extend our former approach for modelling the dependency of a component’s performance to external services [3]. We also implicitly consider influences of different usage profiles by specifying transition probabilities on control flow branches and probability mass functions for the number of loop iterations. We use stochastic regular expressions [4] to construct a performance model based on service effect specifications for a component service [5]. An experimental evaluation is provided, as we have implemented our approach and applied it on an experimental webserver.

Our approach is precise, because we use discrete probability mass functions to model the time consumption of internal and external computations. It is compositional, since the result of our prediction is again a discrete probability mass function and can be used to model the performance of an external service for another component. Furthermore, the approach is parametric, because different probability mass functions can be used (for example depending on the underlying hardware) and the prediction can be adjusted to a different usage profile by changing transition probabilities and probability functions for loop iteration. Moreover, performance contracts for component services specified in the Quality of Service Modelling Language (QML) [6] can be checked with our method.

The contribution of this paper is twofold: First, we present a new concept for modelling loops with probability mass functions and introduce stochastic regular expressions. Second, we report on an experimental evaluation of our approach on a prototypical component-based software system and compare the predictions of
our approach with measurements. As can be seen in our evaluation, the calculated probability mass functions differ only slightly from the actual measured values emphasizing the precision of our approach.

The paper is organised as follows: Section 2 introduces our modelling approach based on service effect specifications, describes the stochastic annotations and explains the calculations. Section 3 contains the description of our experimental evaluation and illustrates the results. Section 4 discusses related work and section 5 concludes the paper and outlines future work.

2 Modeling Component Performance

2.1 Service Effect Specifications

Our model to describe the performance of a component is based on service effect specifications [5], which have been proven to be useful to model the influence of external dependencies on software components. When modelling the performance (i.e. response time, throughput) of a component, the component’s calls to required services have to be taken into account, because they affect the performance perceived by the users of the component. A service effect specification models the calls of a provided service of a component to its required services. Usually, the specification is represented as an automaton to restrict possible sequences of calls. The transitions of the automaton model external calls and the states represent internal computation of the component. Thus, a service effect automaton can be considered as a control flow abstraction.

Fig. 1 shows an example component foo in UML 2.0 notation on the left side, with the provided service z and the required services a, b, and c. The service effect automaton for service z is presented on the right side. It is a finite state machine describing all possible call sequences emitted by z. Service effect specification have to be provided by the component developer for component assemblers. Service effect specifications can be generated out of source code or derived from design documents.
As we also want to model the time consumption of the internal computations of a service, we first decompose the service effect automaton. For each state an additional transition and state is inserted and the transition is labelled with the name of the former state (Fig. 2). For example, a transition $s_2$ is added from the former state $s_2$ to the new final state. This step is also necessary to ease the later computations.

![Decomposed service effect automaton](image)

Loops with the control flow are modelled by cycles in the service effect automaton. Cycles complicate the analysis, since they can be interconnected, include other cycles or have multiple entry points. We want to model loop iterations with probability functions, which will be described later, so we have to identify all cycles explicitly. Because of this, we convert the service effect automaton into a regular expression. Regular expressions are hierarchically structured and loops have a clear entry and exit point and can be identified by the Kleene star operator. Furthermore, regular expressions are well suited for the later computations, because it is possible to perform the calculation by traversing their abstract syntax tree. For the conversion, we use the GNFA-algorithm [7]. The regular expression for the above service effect automaton is:

$$(s_0 \ a \ s_1 \ b)^* \ s_0 \ c \ s_2$$

### 2.2 Stochastical Annotations

Service effect specifications were originally designed to calculate the requires protocol of a software component out of the provides protocol and to support interoperability checking between components [8]. To conduct performance analysis we need probabilities for the branches in the control flow, descriptions on the number of loop iterations and timing values attached to the elements of the regular expressions.

We describe stochastic regular expressions (SRE) (similar to [4]) in the following. Let $a$ be a terminal symbol from an alphabet $\Sigma$. The syntax of a SRE $R$ is recursively defined as:
For the alternative $R_1^{p_1} + \ldots + R_n^{p_n}$, it must hold that $\sum_{i=1}^{n} p_i = 1$, the sum of probabilities for the different alternatives at a branch in the control flow always equals one. The dot within a concatenation can be omitted.

For the loop construct $R^l$, $l$ is defined as a probability mass function over the number of loop iterations $n$ with

$$l : n \mapsto p_n, n \in \mathbb{N}_0, 0 \leq p_n \leq 1, \exists N \in \mathbb{N}_0 : \forall i > N : l(i) = 0, \sum_{n=0}^{N} l(n) = 1$$

With this definition, we assign a probability to each number of loop iterations (e.g. it is possible to define that a loop is executed 3 times with a probability of 0.2 and 6 times with a probability of 0.8). Furthermore, the number of loop iterations is bounded, as we only allow probabilities greater than 0 for a limited number of iterations. This is a more practical approach to model loops than in classical Markov models [9]. There, loops are modelled with a probability $p$ of re-entering the loop and a probability $1 - p$ of exiting the loop. This binds the number of loop iterations to a geometrical distribution, and it cannot be expressed for example that a loop is executed exactly $n$-times. It can not be expressed that a larger number of loop iterations is executed with a higher probability than a smaller one. But such situations can be found in many applications in practice, where a geometrical distribution on loop iterations is rather an exception. The problem of modelling loops with Markov models has also been stated by Doerner et. al. [10], who tackle the problem with a different approach. Other approaches models loops simply by providing mean iteration numbers [11]. This approach is also not favourable, because in reality, the number of loop iterations may differ heavily based on the usage of the component, and a mean value falls short to model this behaviour.

The **semantics** of the above syntax is defined as follows:

- Symbol $(a^{p_a})$: $a$ is executed with probability $p_a$.
- Alternative $(R_1^{p_1} + R_2^{p_2})$: either $R_1$ is executed with probability $p_1$ or $R_2$ is executed with probability $p_2$. The total probability of the expression is $p_{alt} = p_1 + p_2$.
- Sequence $(R_1^{p_1} \cdot R_2^{p_2})$: first $R_1$ is executed, then $R_2$ is executed. The total probability of the expression is $p_{seq} = p_1 \cdot p_2$.
- Loop $(R^l)$:

$$\begin{cases}
R & \text{is executed with probability } l(1) \\
R \cdot R & \text{is executed with probability } l(2) \\
\vdots & \\
R \cdot \ldots \cdot R & \text{is executed with probability } l(n)
\end{cases}$$
The total probability of the expression is \( \sum_{i=1}^{n} l(i) = 1 \).

For the example regular expression from above, a possible SRE could be

\[
(s_0^{p_0} \ a^{p_a} \ s_1^{p_1} \ b^{p_b})^l \ s_0^{p_0} \ c^{p_c} \ s_2^{p_2},
\]

with \( p_{s_0} = p_a = p_{s_1} = p_b = p_c = p_{s_2} = 1,0 \) and \( l(7) = 1,0 \).

To model the timing behaviour of the expression, a random variable \( X_R \) is assigned to each input symbol \( R \) as described in [3]. The random variable can be described by a probability mass function. It models the time consumption of a called service or an internal computation. In the following, we write \( x_{\alpha}[n] \) for a probability mass function

\[
x_{\alpha} : n \mapsto p_n = P(X = \alpha n)
\]

with sampling rate \( \alpha \). The sampling rate \( \alpha \) is the stepwidth for the intervals in the probability mass function and can be chosen for example as the greatest common divisor (gcd) of the interval’s lengths: \( \alpha = gcd\{\|x_{i-1}, x_i]\}_i=1,2,...,m-1 \).

For example, service \( s_0 \) from the stochastic regular expression above is assigned with a random variable \( X_0 \), whose probability mass function models how long \( s_0 \) is executed. The probability mass function is either the result of a computation, measured, or estimated (cf. section 2.4).

Using random variables instead of constant values allows a more fine grain performance prediction and is well suited to model internal and external time consumptions which are not fixed to constant values. They depend on a number of factors like the underlying hardware and middleware platform, the internal state of the component or inputs entered by users.

### 2.3 Computations

The performance of a provided service is computed out of the annotated service effect specifications expressed as a stochastic regular expression. For the computations the abstract syntax tree of the regular expression is built and annotated with the probabilities, loop iterations functions, and random variables for time consumption. The tree is then traversed bottom-up until the resulting random variable for the root node (i.e. for the provided service) is computed. In the following, we explain the computation steps for the basic constructs sequence, alternative, and loop.

The time consumption for a sequence is the sum of the time consumptions for each expression. The probability mass function for a sequence can be computed as the convolution of the single probability mass functions:

\[
x_{R_1 \cdot R_2}(n) = x_{R_1} \circledast x_{R_2}[n]
\]

For an alternative, the time consumption is computed as the sum of the alternative paths weighted by the branch probabilities. The corresponding probability mass function is:
As we have specified a probability mass function for the number of loop iterations, the random variable for the loop is:

\[
x_{R_1+R_2}(n) = p_1x_{R_1}[n] + p_2x_{R_2}[n]
\]

To compute the convolutions of the probability mass functions we make use of discrete Fourier transform as described in [3], where also the computational complexity of the approach is discussed in detail.

The probability mass function for the provided service \( z \) of the component \( \text{foo} \) from the example above has the form:

\[
x_z(n) = \left( \sum_{i=1}^{7} l(i) \right) \otimes x_z[n]
\]

2.4 Getting the necessary values

To conduct the performance analysis with our approach, different inputs are needed. Service effect specifications can be obtained by analysing source code or design documents. For example, UML sequence diagrams might have been specified for some components and contain the information to construct a service effect specification.

The service effect specifications are annotated with transition probabilities and random variables for loop iterations. The component developer cannot provide these values by analysing the code of a component, because these values may depend on how users execute the components. For example, at an alternative in the service effect specification the control flow may take a different direction depending on the input parameter values of the service. Because of this, the values have to be provided by the system assembler, who has a certain expectation on how the anticipated users will use the component. An approach similar to the work by Hamlet et. al. [12] might be taken by the system assembler to obtain transition probabilities and loop iteration probabilities.
Furthermore, the service effect specification is annotated with random variables expressing the time consumption of external services (on transitions) and the time consumption of internal computations (on states). The needed probability functions might be derived from measurements (e.g. by benchmarking the component and the external services) or by estimating the values based on former experience, possibly supported by an approach like SPE [13].

It is still unclear, if our approach can be fully applied on existing black box components, for example by analysing byte code. Code annotations may have to be made to fully retrieve the needed information for our approach from source code. The direction of reengineering existing components to make them useful for our approach is part of our future research.

3 Experimental Validation

In the following, we provide a first initial experimental evaluation of our performance prediction approach. We compare measured data with calculated data assuming that the input data for the calculations are available. This assumption also needs to be tested in the future, as it is not clear if developers of component-based systems can obtain all the necessary data with the needed precision. The assumption could be tested with a controlled experiment involving students or with an experiment in an industrial setting, but this is beyond the scope of the validation described here.

The goal of our experimental evaluation was to analyse our performance prediction method from the viewpoint of the developer. We ask the following questions and use the described metrics to answers these questions:

• **Precision:** How precise are the calculation opposed to the measurements? As metrics we use descriptive statistics.

• **Sampling Rate:** What is the influence of the sampling rate of the probability mass functions to our calculations? We will use descriptive statistics and the time used for the calculations for different sampling rates to answer this question.

• **Efficiency:** How efficient can the calculation be performed? The time for the calculations will be used as a metric here. Furthermore, we will discuss the complexity of the calculations.

3.1 Subject of the Experiment: Webserver

For our experiment, we applied the performance prediction approach on a component-based web server, which has been developed in our group for research purposes. The server has been designed for the .NET-platform and is implemented in C#. It is multi-threaded, as a dispatcher component spawns a new thread for each incoming request. Pages can be served either statically from harddisk, or they can be assembled dynamically by using the contents of a connected database (Microsoft SQL-server).
The webserver consists of 12 components, which are organised in a flexible architecture. Multiple components exist to handle different kinds of requests (e.g. HTTP requests) to the webserver. These components are organised in a Chain-of-Responsibility pattern [14], to easily allow extensions for other kinds of requests (e.g. FTP or HTTPS requests). We omit the full architecture of the server here and focus instead on predicting the performance of a single component, while taking connected components into account.

The StaticFileProvider component (Fig.3) provides an interface called IHTTPRequestProcessor to handle simple HTTP requests and retrieves files from hard-disk. It requires the IWebserverMonitor and the IWebserverConfiguration interface to write log messages and to query the global webserver configuration. Furthermore, the IRequestProcessorTools interface is used to open and send files to the client.

```
StaticFileProvider
 IHTTPRequestProcessor ⊳ IWebserverMonitor ⊳ IWebserverConfiguration ⊳ IRequestProcessorTools
```

Fig. 3. Component StaticFileProvider

The service effect automaton [15] of the service HandleRequest describes the abstract behaviour of the service and contains a transition for each external service called (Fig. 4). First, the path information is built for the requested file and a check is performed if the path actually exists. If a file is requested, the service writes a log entry and then starts to read the file from harddisk and transfer it to the client. If a directory is requested, the service first checks the webserver configuration (via the IWebserverConfiguration interface) for allowed standard file names (e.g. index.htm) and then checks in a loop, if one of the standard file names can be found in the requested directory. This file is retrieved and sent to the client. If the file can not be found or the default file name does not exist, an error message is sent to the client.

The regular expression corresponding to the service effect specification (Fig. 5) has been obtained by applying the GNFA algorithm [7]. The names of the transitions are abbreviated with the capital letter (e.g. BCP for BuildCompletePath). We omit the regular expression for the states of the service effect automaton here for the sake of brevity.

In the following, we define the independent, dependent and possible interfering variables of our experiment. Independent variables are not altered between measurements and calculations and define the context of our experiment. They are namely the hardware, the middleware platform, the deployment, the analysed component (StaticFileProvider) and the usage profile. To reduce the influence of the usage profile on the outcome of the experiment, we analysed three different scenarios for the service effect specification described above, which will be described later.
The dependent variable of our experiment is the metric we measured on the webserver, which was the response time for the different scenarios in milliseconds. Possible interfering variables were active background processes, which could have distorted the measurements, caching of the webserver, time consumption for the output of log messages, the possible influence on the measurement approach to the measured results, garbage collection, just-in-time compiling etc. We tried to keep the effect of the interfering variables as low as possible, by e.g. deactivating background processes, and implementing the measurement application as efficient as possible.
3.2 Scenarios

We applied our method on three different scenarios for the HandleRequest service of the StaticFileProvider component. The scenarios reflect the basic control flow constructs of sequence, alternative, and loop.

In Scenario 1, a 50 KByte HTML-file was requested from the webserver. As no directory was requested, no loop was executed in this scenario ($l_1(i) = 0$, $\forall i \in \mathbb{N}$) and the control flow simply followed a sequence. Fig. 6 shows the stochastic regular expression for this scenario. Included are only the probabilities for alternatives to make the expression more readable, the probabilities for the other expression are all 1.0.

$$
(BCP\ SHE1)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.5} + (BCP\ WL1)^{0.5}) \ (WL3)^{l_1} \ SHE2)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.0} + (BCP\ WL1)^{1.0}) \ (WL3)^{l_1} GFM\ WL4\ OF\ SHH\ SCD\ WL5)^{1.0}
$$

Fig. 6. Stochastic Regular Expression for Scenario 1 (Sequence)

Scenario 2 involved two different request, triggering an alternative in the control flow. A 50 KByte HTML file and a subdirectory were requested alternately. The loop was not executed ($l_2(0) = 1.0$ and $l_2(i) = 0$, $\forall i \in \mathbb{N}$) as the file in the subdirectory was immediately found in this scenario (Fig. 7).

$$
(BCP\ SHE1)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.5} + (BCP\ WL1)^{0.5}) \ (WL3)^{l_2} \ SHE2)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.5} + (BCP\ WL1)^{0.5}) \ (WL3)^{l_2} GFM\ WL4\ OF\ SHH\ SCD\ WL5)^{1.0}
$$

Fig. 7. Stochastic Regular Expression for Scenario 2 (Alternative)

Scenario 3 contained the execution of the loop for 5 times, as a subdirectory was requested, and the webserver searched for 5 different file names before finding the file. Thus, the probability function for the loop iteration was $l_3(5) = 1.0$ and $l_3(i) = 0$, $\forall i \in \mathbb{N}_0 \setminus \{5\}$. Otherwise, the transitions were taken sequentially in this scenario, illustrated by Fig. 8.

$$
(BCP\ SHE1)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.5} + (BCP\ WL1)^{0.5}) \ (WL3)^{l_3} \ SHE2)^{0.0} +\\
((BCP\ GDF\ WL2)^{0.0} + (BCP\ WL1)^{1.0}) \ (WL3)^{l_3} GFM\ WL4\ OF\ SHH\ SCD\ WL5)^{1.0}
$$

Fig. 8. Stochastic Regular Expression for Scenario 3 (Loop)

3.3 Measurements and Calculations

For measuring the scenarios, we implemented a monitoring framework for our webserver, using interceptors to decorate component interfaces with a measuring facility [14]. The response time of each service call was measured. The data was stored in memory during the measuring process and written to disk in an XML file format after the webserver was shut down. This way, we tried to remove the interfering influence of monitoring on the measurement results because of harddisk accesses.
The measurements were performed locally on a laptop with a Pentium-M processor 1.6 GHz, and 512 MB RAM with the webserver running on the .NET-platform. The requests for the scenarios were generated by a commercial web stress testing tool and subsequently repeated for one minute in each scenario. All requests were performed non-concurrently.

For the calculations, we used the measured data for single services as input to specify the random variables of time consumption for each service. The transition probabilities and loop iteration number were generated out of the measured data to ensure consistency between both approaches. During calculations, the abstract syntax tree of the regular expression is traversed and the time consumption of sequences, alternatives, and loops are computed bottom-up.

We used a sampling rate of 10 for each calculation, meaning that 10 values were combined from the probability mass functions to compute each value of the resulting function. Furthermore, we measured the duration for each calculation.

3.4 Results

3.4.1 Precision

Fig. 9-11 show the probability mass functions for the response time (in $\mu$s) of each scenario of the HandleRequest service from the StaticFileProvider component. Measured data (dashed line) is compared with calculated data (dark line).

For the sequential execution of scenario 1, the calculated probabilities closely conform to the measured probabilities (observable in Fig. 9). The calculated line appears more smooth than the measured line because of the convolutions involved in the calculations. As a goodness of fitness test of two distribution functions, a $\chi^2$-test is commonly used in statistics [16]. The $\chi^2$-test with three degrees of freedom on the two distribution functions here yielded a too high value for $\chi^2$ to hold our defined significance level of 0.05. Thus, the deviation of both results was too high to confirm the same underlying probability function.

However, we would still argue that the results are useful because the deviation is small from a developer's perspective. The mean values of measurements and calculations only deviate by 0.45% and the standard deviation by 9.67% (Tab. 1). The maximum values of the probabilities are at 1650 $\mu$s for the measurements and at 1670 $\mu$s for the calculations, meaning that the most probable values only deviate by 1.2 percent in the response time. We also found that 80 percent of the values lie between 1500 and 2000 $\mu$s, both for measurements and calculation. For a performance analyst trying to predict the response time of components during early development stages with lots of still unstable informations, the precision should be adequate to make rough estimations and to support design decisions.

Scenario 2 involved an alternative in the control flow of the service effect specification. Like the sequential execution of scenario 1, the calculated probabilities resemble the measured probabilities closely, as can be observed in Fig. 10. Nevertheless, the hypothesis for the $\chi^2$-test had to be rejected as in scenario 1.

But like in scenario 1, the mean values and the standard deviation of measure-
Fig. 9. Scenario 1 (Sequence): Probability Mass Functions

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Max Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement (μs)</td>
<td>2124,78</td>
<td>1387,57</td>
<td>1650</td>
</tr>
<tr>
<td>Calculation (μs)</td>
<td>2115,15</td>
<td>1253,44</td>
<td>1670</td>
</tr>
<tr>
<td>Deviation (%)</td>
<td>0,45</td>
<td>9,67</td>
<td>1,20</td>
</tr>
</tbody>
</table>

Table 1

Scenario 1 (Sequence): Descriptive Statistics

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Max Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement (μs)</td>
<td>2215,38</td>
<td>1681,25</td>
<td>1650</td>
</tr>
<tr>
<td>Calculation (μs)</td>
<td>2206,66</td>
<td>1508,89</td>
<td>1690</td>
</tr>
<tr>
<td>Deviation (%)</td>
<td>0,39</td>
<td>10,25</td>
<td>2,37</td>
</tr>
</tbody>
</table>

Table 2

Scenario 2 (Alternative): Descriptive statistics

The results for scenario 3 (loop iteration) can be found in Fig. 11. The shape and calculations are very similar (Tab. 2). The maximum probabilities can be found at 1650 μs (measurements) and 1690 μs (calculations), yielding a deviation of 2.3 percent in response time. 75 percent of the probabilities of measurements and calculations lie between 1550 and 1920 μs. So the calculations are almost as precise as in scenario 1.
of the lines of measured and calculated probabilities appear similar, although a deviation can be detected as the measured curve is slightly shifted to the right. As above, the $\chi^2$-test lead to a rejection of the hypothesis of the same underlying probability function.

Fig. 11. Scenario 3 (Loop): Probability Mass Functions
The mean values for measurements and calculations only deviate by 1.58%, the standard deviation and the response time with the highest probability are very similar for both measurements and calculations (Tab. 3). More than 81 percent of the probability values can be found between 6400 µs and 8800 µs for the measured values as also for the calculated values. The precision appears useful for performance predictions again.

In Fig. 11, we have also included the calculations for this scenario with a Markovian loop concept (dotted line). With this concept, probabilities are specified for re-entering and for exiting the loop. The situation found in scenario 3 of exactly five loop iterations cannot be expressed. As a result, the calculation does not resemble the measured values as closely as with our calculations with non-Markovian loop iterations. The curve for the Markovian loop model almost looks like an exponential distribution shifted to the right. The reason for this is, that by specifying loops with transition probabilities, the number of loop iterations is always bound to a geometrical distribution (the discrete case of the exponential distribution). The practical advantage of modelling loops in a non-Markovian way can be clearly observed in the graph, because the predictions do resemble the measurements closer.

### Table 3

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Max Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement (µs)</td>
<td>8160,87</td>
<td>2726,85</td>
<td>6650</td>
</tr>
<tr>
<td>Calculation (µs)</td>
<td>8031,65</td>
<td>2918,98</td>
<td>6850</td>
</tr>
<tr>
<td>Deviation (%)</td>
<td>1,58</td>
<td>7,05</td>
<td>2,92</td>
</tr>
</tbody>
</table>

### 3.4.2 Sampling Rate

To analyse the influence of the sampling rate α from the probability mass functions, we obtained measurements and calculations for scenario 1 with different sampling rates (1, 5, 10, 50). The time consumption for the calculations with the different sampling rates was 103 sec, 7 sec, 3 sec, 2 sec respectively, clearly showing the impact of the sampling rate on the timing efficiency of our approach. Looking at the results (Fig. 12, upper-left: α = 1, upper-right α = 5, lower-left: α = 10, lower right α = 50, it can be observed that the calculations closely resemble the measurements up to a sampling rate of 10. Only the results with a sampling rate of 50 differed from the measurements to a greater extent. This shows, that we can perform our calculations with a higher sampling rate without loosing much precision.

### 3.4.3 Complexity and Efficiency

In the following, we discuss the time complexity and time efficiency of our computations. For calculating the timing behaviour of sequences or loops, convolutions...
of the corresponding probability mass functions have to be performed. We use discrete Fourier transformations, because the convolution becomes a product in the frequency domain.

First we will analyse the time complexity of the calculations. Without loss of generality, consider a random variable $X$ describing the timing behaviour of the loop body. Let $w$ be the number of values of $X$ and let $N$ be the maximal possible number of loop iterations determined by the function $l$. Before the Fourier transformation, the value range of $X$ has to be enlarged to $Nw$. The discrete Fourier transformation of a random variable with $Nw$ values has the complexity of $O(N^2w^2 + Nw)$. The $N$-fold convolution of the discrete random variable corresponds to the $N$-fold pointwise product of the Fourier transform. As the Fourier transform also has $Nw$ values, the complexity of the $N$-fold product is $O(N^2w)$. Afterwards, the inverse Fourier transformation has to be performed, having the same complexity as the Fourier transformation. Altogether, the complexity of the computation of the probability mass function of $N$-fold loop iterations is $O(N^2w^2 + N^2w + Nw)$.

We also took the time for the calculations of the scenarios described above to analyse the actual time efficiency of our approach in our setting. With a sampling rate of 10, the calculations consumed 3 seconds for scenario 1, 7 seconds for scenario 2, and 3 seconds for scenario 3. Additional time was consumed to derive the input data for the calculations from the measurements. For our evaluation, the time efficiency of our calculations was adequate. So far, we have not optimised the code of the calculations, neither have we analysed more complex components.
4 Related Work

Our approach aims at supporting design decisions during early life-cycle stages of a software system. The timing and resource efficiency of a component-based software architecture shall be assessed as early as possible during development to avoid the costs of redesign after starting the implementation. The SPE (Software Performance Engineering) methodology by Smith et. al. [17,13] was one of the first approaches into this direction. Balsamo et. al. [18] provide a broad survey on model-based performance prediction methods.

Recently, Becker et. al. [2] specifically classify performance prediction methods for component-based systems. They distinguish between quantitative and qualitative approaches. Quantitative approaches are refined into measurement-based, model-based, and combined approaches. Our performance prediction method is a quantitative approach, because it aims at providing the system designer with performance metrics such as response time or throughput. It is furthermore model-based, as a special kind of stochastic process is used to carry out the analysis.

A number of other model-based approaches have been proposed. Sitaraman et. al. [19,20] tackle the difficult problem of specifying the performance of a software component. They extend classical O-Notations to specify the time and space requirements of a component and also address the dependency of performance to input data. Hissam et. al. [21] propose a common terminology for Quality of Service prediction of component-based systems. Hamlet et. al. [12] use the information of how subdomains of inputs on provided interfaces are mapped to the interfaces of subsequent components to make performance and reliability predictions. Bertolino et. al. [11] developed the CB-SPE framework, which is based on the SPE-methodology. In this approach, UML models for component-based architectures are annotated with performance values according to the UML SPT profile [22] and then mapped to queueing networks. Wu et. al. [23] define an XML-based Component-Based Modelling Language to describe the performance of a component-based software system and generate Layered Queueing Networks. Eskenazi et. al. [24] introduce an incremental method to predict the performance of system composed out of components, allowing analytical, statistical, or simulation-based models at each step of the approach. The approach by Chaudron et. al. [25] aims at real-time component-based systems and uses simulations to predict the performance.

Further work is related in terms of the notations used. Parametric contracts for software components have been developed by Reussner et. al. [15]. Stochastically enhanced service effect specifications have been used for reliability predictions for component based architectures [26]. The Quality of Service Modelling Language (QML) [6] can be used to express performance contracts for components. Stocharts [27] are stochastically enhanced statecharts similar to our annotated service effect specifications. However, the number of loop iterations is bound to a geometrical distribution in this approach. The problem of modelling loops with Markov chains is discussed by Doerner et. al. [10]. Stochastical regular expressions have
been used for example by Garg et. al. [4]. The SOFA component model uses
behavioural protocols similar to our provides and requires protocols to specify al-
lowed sequences of services calls, which are used for interoperability checking
[28].

5 Conclusions and Future Work

In this paper, we have extended our previous parametric performance contracts [3]
by introducing a new loop concept to better model practical situations. We have
used stochastic regular expressions instead of Markov models, because it is easier
to identify loops and the effort for the calculations is reduced by the ability to
traverse the abstract syntax tree of the expressions. Furthermore, we presented an
experimental evaluation, finding a close resemblance between measured data and
data calculated. We also discussed the efficiency of the approach.

Encouraged by our experimental evaluation, we think that our approach of mod-
elling component performance will be useful for component developers and system
assemblers in the future. Architects of component-based systems will be able to
identify bottlenecks in their architectures and will also be supported when choosing
among components with equal functionality but different QoS-characteristics.

However, the approach is far from being complete or being applicable in prac-
tice right away. Future work aims at including concurrency to be able to model a
large class of practical applications. Furthermore, the performance predictions shall
be parameterised for the underlying hardware and middleware platform, which is
still implicit in our probability function. We also envision a better treatment of the
usage profile of components, as so far we assume that the necessary values for us-
age modeling are already available. The computational complexity as well as the
practical feasibility also have to be analysed more in depth.4

Acknowledgements: We would like to thank Jens Happe and Helge Hartmann
for helping us to implement and validate our approach.

References

[1] Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented
Programming. Addison-Wesley (2002)

[2] Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of
component-based systems: A survey from an engineering perspective. In Reussner, R.,
Number To Appear in LNCS. Springer (2005)

Further details on the Palladio project are available at:
http://se.informatik.uni-oldenburg.de/palladio

82


83


A Contract-based Approach to Specifying and Verifying Safety Critical Systems

Wei Dong, Zhenbang Chen and Ji Wang

National Laboratory for Parallel and Distributed Processing
ChangSha, P.R.China
{dong.wei, z.b.chen, jiwang}@mail.edu.cn

Abstract
Light-weight formal method has been regarded as an important approach to development of component-based safety critical systems. The paper proposes an approach which can formally specify and verify the contract of static structure, dynamic behavior and refinement of component systems based on UML 2.0 superstructure. As results, the correctness of static contract can be obtained via type checking of interfaces and connectors. Dynamic contract can be verified through determining the cooperativeness of integrated components, whose contracts are depicted with interface protocol state machines and their semantics models, namely contract automata. The refinement relation between high level component and its implementation will be guaranteed through defining the alternating simulation between contract automata of components at different levels.

Key words: formal specification, software verification, component-based software development

1 Introduction
It is an admitted fact that software has become the pivotal element in safety critical systems, such as avionics and aerospace systems, nuclear power controller, etc. Generally, these systems have the characteristics of real-time, dynamic, autonomy, fault-tolerant, and should satisfy some critical properties. Because of the complexity and demand of high confidence, the development, deployment and running of these systems are faced with grand challenges.
Component-based software development (CBSD) can effectively deal with the complexity of software construction, and provides the better manner for describing architectures which present systematic frameworks of applications in specific domains. The paradigm of CBSD has also been recognized in developing safety critical systems[1], in which it is convinced that the components with dependable guarantee will be the basis, and the corresponding methodology for component integration, analysis and verification should be studied. Formal method makes it possible to calculate whether a certain description of a system is internally consistent, or whether requirements have been satisfied in the derivation of a design. Formal method has been widely accepted in developing safety critical systems. Under CBSD, formal method calls for the appropriate techniques of formal specification, integration, analysis and verification around safety and reliability properties.

This paper studies the formal specification and verification of both static and dynamic contract in developing component-based safety critical systems. Its background is the development of SAFE-II, the lifesaving system of manned spaceship, which uses CBSD as the paradigm. It is expected that the resulting formalism will be suitable for formal analysis and verification of SAFE-II, and the other similar systems as well. After we have investigated the characteristics and development demands of these systems, several principles will be followed in our study.

(1) For the sake of practicability in component-based design and construction, a popular opinion is that the light-weight formal method should be considered under mainstream methodology and modelling language.

(2) To verify essential properties of safety critical systems (e.g. via model checking), the dynamic or temporal aspects of interfaces should be specified in the contract. Moreover, the dynamic contract should support the component composition and refinement, which are common activities in CBSD.

(3) The architecture pattern should be considered to improve the effectiveness of formal specification and verification. For example, in embedded safety critical systems, the interface usages of software components are usually periodical and time-triggered[2].

(4) The formalism should be able to describe the compatibility between component system and the environment, or distinguish which are legal environments for the system.

Unified Modeling Language (UML) provides various viewpoints and diagrams to depict the characteristics of software systems. Comparing with former versions, UML 2.0 has distinctly improved the descriptive capability of component models[3]. This paper firstly formalizes the component model of UML 2.0, which will be used to model the static structures of safety critical systems. To depict the temporal constraints of interface usages, the dynamic light-weight formal specifications are attached to the components through defining interface protocol state machines (IPSM) and their seman-
tics models, namely contract automata. The notions of stateful and stateless are introduced into contracts to distinguish the specialties of services.

Time-triggered mode has been widely adopted in safety critical community because it is somewhat predictable, and can reduce the complexity and improve the reliability and safety[2,4]. SAFE-II also can be regarded as the time-triggered system. Therefore, the time-triggered pattern is brought into the component composition (But in current formalization, the real-time properties are not considered yet). Then the essential static and dynamic consistency rules for component integration are studied respectively. By the rules, the static contract can be verified via type checking of interfaces and connectors, and the dynamic contract can be verified by investigating if the integrated components are cooperative. Based on dynamic contract, the way of how to specify and determine the legal environments is presented. The refinement relation between high level component and its implementation is also studied, and the refinement consistency can be checked for both top-down design and bottom-up construction.

The method has been practically applied in SAFE-II, and it is also applicable for other safety critical systems, such as aircraft autopilot or train control systems[5]. These formal specifications and contracts are also the foundation of component-based system verification in our future work, such as model checking, compositional reasoning, and real-time architecture development.

The next section discusses the related work. The formal specification of component system structure is given is section 3, and the static consistencies are proposed in section 4. The dynamic contract and consistency are studied in section 5, and the refinement relation among components is studied in section 6. The last section concludes the paper with the future work.

2 Related Work

There has been some work related to applying formal methods for developing software architectures and component-based systems. Wright[6] and Darwin[7] are typical ones of early architecture description languages. Wright defines the possible interactions between a set of roles, whose behaviors are specified in CSP. Darwin describes software architecture with π-calculus. In SOFA[8], an application is viewed as a hierarchy of nested software components which can be deployed over a network. The behavior of a component in SOFA is approximated by a regular language which can be expressed by a behavior protocol, whose conformance is also defined. Archware[9] provides a style-based executable language which is a framework for formalizing architectures based on components and connectors. It can be used to describe architectural structure, behavior, qualities and evolution of systems. [10] presents a method for compositional verification of middleware-based software architecture. It is a framework not related to any specific specification of component system. In formalization of dynamic contract, interface automata[11] are similar to our
approach. But interface automata don’t consider the structure of component system as well as concurrency and hierarchy of dynamic contract, and use the optimistic approach in composition. Without consideration of stateless interfaces, an interface explicitly only can be used by at most one component in composition of interface automata.

Some other work is more closely related to specification and verification of component-based embedded or safety critical systems. [12] applies concurrency controller design pattern in Tactical Separation Assisted Flight Environment(TSAFE), and separates behavior verification from interface verification. Cadena[13] is an integrated environment for building and modelling systems using CORBA Component Model(CCM). The model can be translated into the input language of DSpin for model checking. [14] also studied the model checking of component composition, in which the behavior of each component is represented with Moore state model. SaveCCM[15] is a component model for safety critical real-time systems, and it is a part of component technology which is intended to provide efficient development, predictable behavior, and run-time efficiency. But SaveCCM has not been formalized. [16] and [17] give two different approaches in defining behavioral models of components and composition. The former one considers the hierarchies, which are not included in the latter. None of them specify the refinement relation.

All the above work can not fully meet the 4 principles proposed in section 1, and may not suit UML 2.0 well. Approach presented in this paper attempts to form a light-weight specification and verification method for both composition and refinement of UML 2.0 component model.

3 Component Model Based on UML 2.0

In component model of UML 2.0, provided or required interfaces are offered through ports, and an interface can contain more than one operations. Components are integrated through assembly and delegation connectors[3]. For example, the fault-detecting component in SAFE-II is presented in Fig.1.

**Definition 3.1 (Interface)** An interface $u$ is defined as the tuple $(O, type)$, in which $O$ is the operation set $\{op_1, op_2, ...\}$, and $type$ can be provided or required. For clarity, $u!$ or $u?$ can be used to denote that an interface $u$ is a provided or required interface.

Let $\overline{u!} = u?$, $\overline{u?} = u!$. For simplification in the following, an operation $op$ of provided(required) interface can also be called provided(required) operation and written as $op!(or \ op?)$.

**Definition 3.2 (Component)** A component $C$ is a 6-tuple $(I_P, I_R, P, R, f_C, G)$, in which $I_P$ and $I_R$ are provided and required interface set respectively. $P$ is the set of ports. $R \subseteq P \times (I_P \cup I_R)$ maps the ports to interfaces. $f_C : \bigcup_{u \in I_P} u.O \rightarrow \{true, false\}$ is a boolean function. $G$ is the sub-component diagram of $C$. 

88
An operation $op$ is stateful if $f_C(op) = true$, otherwise it is a stateless operation. $f_C$ will be used in dynamic contract. The component is a basic component if $G$ (see definition 3.5) is empty, otherwise a composite component. The set $I_P$ and $I_R$ form the external view of the component, and $G$ describes the internal view. Assembly and delegation connectors will be used to construct the component diagram and map one component’s external view to internal view. Port can be regarded as the bridge in delegation.

**Definition 3.3 (Assembly Connector)** An assembly connector is denoted as $(u!, v?)$, which connects a provided interface to a required interface.

**Definition 3.4 (Delegation Connector)**

1) A provided delegation connector $(p, u!)$ delegates the external provided interface linked with port $p$ to provided interface $u!$ of internal view.

2) A required delegation connector $(u?, p)$ delegates the service required by interface $u?$ to external required interface linked with port $p$.

For example, $(AO2!, EI?)$, $(P4, EO!)$ and $(AI1?, P1)$ in Fig.1 are assembly, provided delegation and required delegation connectors respectively. A component diagram describes how the components are connected and form the composite component or system.

**Definition 3.5 (Component Diagram)** A component diagram $G$ is a 3-tuple $(V, F, E)$. $V$ is the component set. $F$ is a set of at most one component, which is the parent of components in $V$. $E = (N_A, N_{D_p}, N_{D_r})$. $N_A$ is the set of assembly connectors, and for $\forall (u!, v?) \in N_A$, $\exists C_1, C_2 \in V$, $u! \in C_1.I_P$, $v? \in C_2.I_R$. $N_{D_p}$ and $N_{D_r}$ are provided and required delegation connector set respectively:

(i) If $F \neq \emptyset$, i.e. $G$ is the sub-component diagram of the only component $C = (I_P, I_R, P, R, f_C, G) \in F$:

(a) For $\forall (p, u!) \in N_{D_p}$, $\exists C_1 \in V$, $u! \in C_1.I_P$, and $p \in C.P$.

(b) For $\forall (u?, p) \in N_{D_r}$, $\exists C_1 \in V$, $u? \in C_1.I_R$, and $p \in C.P$.

(ii) If $F = \emptyset$, i.e. $G$ is a topmost component diagram which does not belong to any component, $N_{D_p} = \emptyset \land N_{D_r} = \emptyset$. 

![Fig. 1. Fault-detecting component in SAFE-II](image-url)
In a component diagram $G$, components in the same hierarchy are connected through assembly connectors. Some interfaces of these components can be connected with the ports of the component which regards $G$ as sub-component diagram. In Fig.1, the elements inside $Fault_Detection$ form its sub-component diagram. Every component in diagram can has its own sub-component diagram, thus hierarchy model will be obtained.

4 Static Consistencies

In component integration, the connected interfaces should be compatible to ensure correct mapping between requests and services. These are static constraints which can be verified via type checking. Some static consistencies that assembly and delegation connectors should satisfy are defined in the following.

**Consistency 1** An assembly connector $(u!, v?)$ is consistent if $v.O \subseteq u.O$.

This rule requires that for an assembly connector, all operations needed in required interface should be served by the provided interface.

**Consistency 2** For a component $C = (I_P, I_R, P, R, f_C, G)$ and a port $p \in P$, if $R(p) \in I_P$ and $G$ is not empty, let $G = (V, F, E)$, then for all $(p, u_i!) \in E.N_{D_p}$, $R(p).O \subseteq \cup u_i.O$.

For example, in the left of Fig.2, interfaces $Prov2!$ and $Prov3!$ must implement all the services that can be provided by $Prov1!$.

**Consistency 3** For a component $C = (I_P, I_R, P, R, f_C, G)$ and a port $p \in P$, if $R(p) \in I_R$ and $G$ is not empty, let $G = (V, F, E)$, then for all $(u_i?, p) \in E.N_{D_p}$, $\cup u_i.O \subseteq R(p).O$.

For example, in the right of Fig.2, the requirements defined in $Req1?$ must contain all the required operations of $Req2?$ and $Req3?$.

5 Dynamic Contract

Safety critical systems have rigorous requirements in temporal order of interface operation usage. Thus, it is necessary to define the dynamic contracts for components. But it is not for all interfaces that should have temporal constraints. Some provided operations can offer services at any time, not only in specialized states. Therefore, provided operations will be classified into
stateful and stateless operations.

We define the interface protocol state machine (IPSM), which is composed of protocol states and protocol transitions, to depict dynamic usage of stateful operations. Protocol state represents an observable stable situation, and the label of protocol transition is an operation of provided or required interface.

**Definition 5.1 (Interface Protocol State Machine)** IPSM is a 7-tuple \((S, L, T, H, K, I, F)\), in which \(S\) is the state set, set \(L\) contains the labels which are interface operations, \(T \subseteq S \times L \times S\) is the transition set. \(H : S \rightarrow \{AND, OR, BASIC\}\) identifies the types of states. \(K : S \rightarrow S \cup \{ROOT\}\) returns the parent state of each state in \(S\), and the parent of topmost states in IPSM is \(ROOT\). \(I\) and \(F\) are initial and final state set respectively.

The IPSM of a component \(A\) is denoted as \(M_A\). The \(AND\) and \(OR\) states of IPSM are similar to that in UML Statecharts\[3\] which model the concurrency and hierarchy structures. But the transitions in IPSM are not event-triggered. Any protocol transition labelled by a required operation means that if the component locates in the source state, it will require the service from other components. A transition labelled by a provided operation means there should exist another component requiring the operation. The IPSM of sub-component \(Arg_{\_\_Reader}\) in Fig.1 is depicted in Fig.3. For clarity in the following discussion, each basic state and operation is assigned a distinct number or abbreviation(e.g., \(DS_{\_Ready}\) is assigned 1, \(AI2?_{\_DigitalSignal}\) is abbreviated as \(op1?\)).

For an operation \(op!\) of component \(A\), \(f_A(op!) = true\) if it appears in some transition label of \(M_A\), else \(f_A(op!) = false\). It is assumed that a stateful operation can not be used as the stateless operation simultaneously. If a provided operation can be used by more than one components, it must be stateless. Two complemented operations may become an internal operation after the component composition.

**Definition 5.2 (Configuration)** A configuration of IPSM is a set \(Conf \subseteq S\) that satisfies: (1) \(\exists s \in Conf\) that \(K(s) = ROOT\); (2) For \(\forall s \in Conf\), if \(H(s) = AND\), for \(\forall s' \in S\) that \(K(s') = s, s' \in Conf\); (3) For \(\forall s \in Conf\), if
Fig. 4. Contract automaton $\Theta_{AR}$ obtained from the IPSM of Arg\textsubscript{Reader}

$H(s) = OR, \exists s' \in Conf$ that $K(s') = s$.

An IPSM must locate in a configuration at any time. Interleaving semantics is considered for AND states according to the characteristic of SAFE-II, and the semantics of entering and exiting composite states is the same as in UML Statecharts. The transition model generated from IPSM $M_A$ is named contract automaton, denoted as $\Theta_A$.

**Definition 5.3 (Contract Automaton)** A contract automaton $\Theta_A$ generated from an IPSM $M_A$ is a 6-tuple $(S_A, s^I_A, O^P_A, O^R_A, O^N_A, T_A)$:

- $S_A$ is the state set (i.e. the configurations of IPSM $M_A$).
- $s^I_A \in S_A$ is the initial state.
- $O^P_A$, $O^R_A$ and $O^N_A$ are set of provided, required and internal operations respectively. Let $O_A = O^P_A \cup O^R_A \cup O^N_A$.
- $T_A \subseteq S_A \times O_A \times S_A$ is the transition set.

The contract automaton generated from IPSM of component Arg\textsubscript{Reader} is presented in Fig.4 (For clarity, only basic states are listed in the configurations). Let $O^P_A(v), O^R_A(v)$ and $O^N_A(v)$ be the set of provided, required and internal operations labelled in the transitions from $v$ respectively, and $O_A(v) = O^P_A(v) \cup O^R_A(v) \cup O^N_A(v)$.

The behavior of composite component can be obtained through the composition of the contract automata of sub-components. SAFE-II can be regarded as the system with time-triggered pattern\cite{2}, for example, the component Fault\_Detection will periodically execute. In the beginning of each cycle, the contract automaton of each sub-component will locate in its initial state. During the execution, the components synchronize through interface invocation. But they should return back to the initial states at the beginning of next cycle. Therefore, in the semantics of composition, both of the synchronization of interface interaction and time-triggered pattern should be included.

Two contract automata $\Theta_A$ and $\Theta_B$ are composable if $O^P_A \cap O^P_B = \emptyset$, which means that the services provided by two components have no overlap.
share\((A, B) = (O_A^P \cap O_B^P) \cup (O_A^R \cap O_B^R)\) includes those operations that are required in one component and provided by the other one.

**Definition 5.4 (Composition)** For two composable contract automata \(\Theta_A\) and \(\Theta_B\), their composition \(\Theta_C = \Theta_A \otimes \Theta_B\) is also a contract automaton:

- \(S_C \subseteq S_A \times S_B\)
- \(s^I_C = (s^I_A, s^I_B)\)
- \(O^P_C = (O^P_A \cup O^P_B) \setminus \{o|o \in share(A, B), f_A(o) \lor f_B(o)\}\)
- \(O^R_C = (O^R_A \cup O^R_B) \setminus share(A, B)\)
- \(O^N_C = O^N_A \cup O^N_B \cup share(A, B)\)
- \(f_C = (f_A \cup f_B)|\{DOM(f_A) \cup DOM(f_B)\} \cap share(A, B)\)
- \(T_C = ((v, u), o, (v', u'))\ in sequence \(v, u, v', u'\) is the first state that \(v_i \neq s^I_A\) and \(u_i \neq s^I_B\), if \(v_j (u_j)\ is the first state after \(v_i (u_i)\) in sequence \(v_0, u_1, ..., v_{i-1}, u_{i-1}\) that \(v_j = s^I_A (u_j = s^I_B)\), \(v_j (u_j)\) can not be left before the sequence \(u_0, u_1, ..., v_{i-1}, u_{i-1}, v_i, u_i\) returns to \(s^I_B (s^I_A)\) after \(u_i (v_i)\).

The computing of \(T_C\) illuminates that the executions of two contract automata are interleaved if the operations labelled on transitions are not included in \(share(A, B)\) or are stateless, and should synchronize if the transitions labelled with stateful operations in \(share(A, B)\) are met. The last item in above definition requires that both of the two contract automata of time-triggered periodic components should return to the initial states before the start of next cycle. The stateless operations can still be used by other components after periodic components should return to the initial states before the start of next cycle. The stateless operations can still be used by other components after periodic components should return to the initial states before the start of next cycle.

Fig.5 gives the IPSM and contract automaton \(\Theta_{FA}\) of \(Fault\_Adjudicator\), which also is the sub-component of \(Fault\_Detection\) and will be integrated with \(Arg\_Reader\) through the composition of \(\Theta_{AR}\) and \(\Theta_{FA}\). For simplifica-
tion, we only consider two paths $\delta_1 : a, b, e, h, i, j, k, a, \ldots \in \Theta_{AR}$ and the path $\delta_2 : l, n, o, p, q, l, \ldots \in \Theta_{FA}$. From the IPSM models, operation op5 is included in $\text{share}(AR, FA)$, because op5 in $\text{Fault\_Adjudicator}$ will be served by op5! in $\text{Arg\_Reader}$. In the composition, the prefix before $i$ in $\delta_1$ and prefix $l, n$ in $\delta_2$ can be interleaved. Then the transition $(i, j)$ in $\delta_1$ and $(n, o)$ in $\delta_2$ should be synchronized. When the composition automaton reaches to state $(j, o)$, $\delta_1$ and $\delta_2$ can interleave again. But when either of two path returns to the initial state, supposing that $\delta_1$ enters $a$, it should wait for the other path returns to the initial state, in this case $\delta_2$ enters $l$.

The synchronization in initial states comes from the requirement in practical implementation, and also can reduce the number of transitions, even states, of the composed system, which is beneficial to formal analysis and verification. It can be proved that the composition is commutative and associative.

**Theorem 5.5** If contract automata $\Theta_A$, $\Theta_B$ and $\Theta_C$ are pairwise composable, $(\Theta_A \otimes \Theta_B) \otimes \Theta_C = \Theta_A \otimes (\Theta_B \otimes \Theta_C)$.

For composable contract automata $\Theta_A$ and $\Theta_B$, there may exist some states in $\Theta_A \otimes \Theta_B$ from which the automaton can not find the next step to execute. Some of these states exist because of the deadlock, and will be put into the set $dl(A, B)$. The other of these states exist because of the synchronization requirement for initial states between two automata in time-triggered pattern. It can be regarded as the pseudo-deadlock, and the states will be put into the set $pdl(A, B)$.

\[
dl(A, B) = \{(v, u) \in S_A \times S_B | (O_A(v) \subseteq \text{share}(A, B)) \land (O_B(u) \subseteq \text{share}(A, B)) \land (\forall o \in O_A(v), \forall o' \in O_B(u), \sigma \neq o') \land (\forall o' \in O_B(u).f_B(o')) \land (\forall o' \in O_B(u).f_B(o'))\}
\]

\[
pdl(A, B) = \{(v, u) \in S_A \times S_B | ((O_A(v) \subseteq \text{share}(A, B)) \land u = s^l_B \land (\forall o \in O_A(v).f_B(o))) \lor (O_B(u) \subseteq \text{share}(A, B)) \land v = s^l_A \land (\forall o \in O_B(u).f_A(o))\}
\]

For example, in Fig.6 (a), when $o_1, o_2 \in \text{share}(A, B)$, $\sigma_1 \neq o_2$, and actually $o_1$ is $o_1!$, or it is $o_1$? but $f_B(o_1!) = \text{true}$, and $o_2$ is $o_2!$, or it is $o_2$? but $f_A(o_2!) = \text{true}$, then the deadlock arises and state $(v, u)$ will be included in $dl(A, B)$.

In Fig.6 (b), when $o \in \text{share}(A, B)$, and if $o$ is actually $o!$, or it is $o$? but $f_A(o!) = \text{true}$, then the state $(v, u)$ will be included in $pdl(A, B)$. The reason is that after $\Theta_A$ and $\Theta_B$ running from the initial states in some cycle, when $\Theta_A$ again returns to initial state $v$, it should wait for the moment that $\Theta_B$ returns to its initial state. But when $\Theta_B$ locates in $u$, it will wait for the synchronization through operation $o$. Then a pseudo-deadlock appears, which is actually a design error for time-triggered pattern.

**Definition 5.6 (Cooperative)** Two contract automata $\Theta_A$ and $\Theta_B$ are cooperative if:

1. $\Theta_A$ and $\Theta_B$ are composable;
2. None of states in $dl(A, B)$ is reachable;
(3) For any path \((v_0, u_0), (v_1, u_1), \ldots\) in \(\Theta_A \otimes \Theta_B\), \((v_i, u_i)\) is the first state that \(v_i \neq s_A^I\) and \(u_i \neq s_B^I\), if \((v_j, u_j)\) is the first state that \(v_j = s_A^I\) and \(u_j = s_B^I\) after \((v_i, u_i)\), then none of states in \(pdl(A, B)\) appears in the sequence segment \((v_i, u_i), \ldots, (v_{j-1}, u_{j-1})\).

When one component is integrated with the others, it is expected that the states in \(dl\) are unreachable at any time, and the states in \(pdl\) are unreachable after each component has left initial state in a cycle.

**Consistency 4** A component diagram \((V, F, E)\) is consistent iff the contract automata of any two components in \(V\) are cooperative.

In practice, it is unnecessary to check each pair of components in verifying this rule. The verification can be combined with the process of composition. Because \(\otimes\) is commutative and associative, it can be proved that if the above consistency is not satisfied, then in any composition sequence \(\Theta_{A_n} \otimes (\Theta_{A_{n-1}} \otimes (\ldots \otimes (\Theta_{A_2} \otimes \Theta_{A_1})\ldots)\) selected to compute \(\otimes V\) (denotes the composition of contract automata of all components in \(V\)), there will exist some \(i\) that \((\Theta_{A_i} \otimes (\Theta_{A_{i-1}} \ldots \otimes (\Theta_{A_2} \otimes \Theta_{A_1})\ldots)\) is not cooperative with \(\Theta_{A_{i+1}}\); on the other side, if the consistency is satisfied, the case of uncooperative will not exist in any composition sequence. Therefore, we can choose any one composition sequence to compute \(\otimes V\), and if an uncooperative case is faced, the consistency is not met, otherwise it is satisfied.

The above consistency can only ensure the correctness among components within the system, but there still may be some required interfaces should be served by the components outside the system, i.e. the environment. To make testing, analysis and verification feasible, the way to specify legal environment with which the component system can work well should be provided.

**Definition 5.7 (Legal Environment)** A Legal environment of a contract automaton \(\Theta_A\) is a nonempty contract automaton \(\Theta_E\) such that:

1. \(O_A^R = O_E^R\);
2. \(\Theta_A\) and \(\Theta_E\) are cooperative.

Fig.7 (a) presents an environment of contract automaton \(\Theta_{FA}\), and their composition is given in Fig.7 (b), from which it can be concluded that this environment is legal for \(\Theta_{FA}\).
Fig. 7. One legal environment of $\Theta_{FA}$ and their composition

For a close system, none external service will be needed, only the consistency 4 should be satisfied. When an open system has been constructed, it should be considered which are legal environments of the system, and whether the system can correctly work in some given environment. It is obvious that an open system can only work well in its legal environments.

6 Component Refinement

In CBSD, a high level component may be refined into an implementation component, or be constructed through integrating more than one sub-components, which can also be regarded as its implementation. The high level component and its implementation must satisfy not only the static consistencies of delegation, but also the consistency between their dynamic behaviors, e.g. IPSMs or contract automata. Hence there should exist a refinement relation between a component and its implementation.

For two components $A$ and $B$, if $B$ refines $A$, it is said $A$ is the specification and $B$ is the implementation. Refinement is traditionally defined as trace containment or simulation. However, it is not appropriate in component systems, and refinement defined through alternating simulation has been proposed[18,11]. In practice, if $B$ implements $A$, $B$ must be able to provide at least all the services defined in $A$. Otherwise, the other components can not find the services they need according to the specification of $A$. On the other hand, $B$ should not require more services provided by other components than that required in $A$, otherwise it perhaps can not work correctly because there may be no component provides such services according to the requirement of $A$. Then it can be concluded that if $B$ refines $A$, $B$ can have more provided interfaces and less required interfaces than $A$. Moreover, the temporal of dynamic behaviors of $B$ should be consistent with that of $A$. In one word, $B$ must be able to work correctly in the environments in which $A$ can work well.

If $A$ is implemented by more than one components $B_i$, $B$ is regarded as the composition of these $B_i$, then $A$ and $B$ should also satisfy the above claim. But for each sub-component, it may has its own interface operations which interact only with other sub-components in the same level. These operations will become the internal operations after integration, and will not affect the refinement relation between $B$ and $A$. We will study the refinement based on
contract automata.

The internal transitions can not be seen by the environment, which leads to the two states connected by an internal transition can not be distinguished. For the simulation focuses on provided and required operations here, a state $s$ and the states reached only by internal transitions from $s$ can be merged.

**Definition 6.1** Given a contract automaton $\Theta_A$ and a state $s \in S_A$, the set $\Gamma_A(s) \subseteq S_A$ is the smallest set such that 1) $s \in \Gamma_A(s)$, 2) if $v \in \Gamma_A(s)$ and $\exists o \in O^N_A, (v, o, u) \in T_A$, then $u \in \Gamma_A(s)$.

Each required operation that may execute in the transition from some state in $\Gamma_A(s)$ should be served by the environment, because any of these transitions may be issued without forewarning. But the environment can only require the service that can execute in some transition from each state in $\Gamma_A(s)$, because it can not distinguish in which state the component is locating. Therefore, some notations will be presented according these facts. Given a contract automaton $\Theta_A$ and a state $s \in S_A$,

$$EX^R_A(s) = \{o | \exists v \in \Gamma_A(s), o \in O^R_A(v)\}$$

$$EX^P_A(s) = \{o | \forall v \in \Gamma_A(s), o \in O^P_A(v)\}$$

For any $o \in EX^P_A(s) \cup EX^R_A(s)$, let

$$Dest_A(s, o) = \{u | \exists (v, o, u) \in T_A, v \in \Gamma_A(s)\}$$

Then the alternating simulation between contract automata can be defined.

**Definition 6.2** Given two contract automata $\Theta_A$ and $\Theta_B$, a binary relation $\succeq \subseteq S_A \times S_B$ is an alternating simulation if for any two states $v \in S_A$ and $u \in S_B$ that $v \succeq u$, the following conditions are satisfied:

1. $EX^P_A(v) \subseteq EX^P_B(u)$, $EX^R_B(u) \subseteq EX^R_A(v)$.
2. For each operation $o \in EX^P_A(v) \cup EX^R_B(u)$ and each state $u' \in Dest_B(u, o)$, there is a state $v' \in Dest_A(v, o)$ such that $v' \succeq u'$.

Two contract automata has the refinement relation if the initial state of first component is alternating simulated by the initial state of second one.

**Definition 6.3** The contract automaton $\Theta_B$ refines the contract automaton $\Theta_A$, written $\Theta_A \succeq \Theta_B$, if 1) $O^P_A \subseteq O^P_B$ and $O^R_B \subseteq O^R_A$, 2) there is an alternating simulation $\succeq$ from $\Theta_B$ to $\Theta_A$, such that $s^I_A \succeq s^I_B$.

It should be guaranteed that the refinement relation must be kept for composition $\otimes$, which will ensure the specification can be safely replaced by its implementation in component integration. The following theorem will ensure the refinement relation can still be hold after composition. The cooperativeness also is kept after the refinement.

**Theorem 6.4** For the contract automata $\Theta_A$, $\Theta_B$, $\Theta_A'$ and $\Theta_B'$ that $\Theta_A \succeq \Theta_A'$, $\Theta_B \succeq \Theta_B'$:
(i) If $\Theta_A$ and $\Theta_B$, $\Theta_{A'}$ and $\Theta_{B'}$ are composable respectively, then $(\Theta_A \otimes \Theta_B) \succeq (\Theta_{A'} \otimes \Theta_{B'})$;

(ii) If $\Theta_A$ and $\Theta_B$ are cooperative, then $\Theta_A'$ and $\Theta_B'$ are cooperative too.

Component-based software development can be considered in two directions: top-down design and bottom-up construction. In both methods, the following refinement consistency should be followed.

Consistency 5 For a component $A = (I_P, I_R, P_R, f_A, G)$ in which $G = (V, F, E)$, if $V \neq \emptyset$, then $\Theta_A \succeq \otimes V$ should be satisfied.

7 Conclusion and Future Work

The industry community has attempted to take the advantages of CBSD paradigm in safety critical system development, in which formal method is desired to be applied. This paper presents our first step to bridge the gap between formal method and the most widely used modeling language UML. The improvement of component model in UML 2.0 also helps us to realize the purpose. We study the characteristics and pattern of component-based safety critical systems like SAFE-II, and propose the formal specification of static structure and dynamic behavior. The consistency rules for static connection, dynamic composition and component refinement are also studied, which can be regarded as one way of verification. The method and supporting tool are being applied in the practical development of SAFE-II. The preliminary results show that they have good usability, and provide a kind of rigorous way to develop component-based safety critical systems.

Around these formal specifications, model checking of component-based safety critical systems is now being studied, especially combined with compositional reasoning to improve the scalability. Timing constraints are unavoidable in these systems (e.g. SAFE-II), and how to introduce real-time model and related performance interfaces into specification and verification will be further studied.

References


Only the Best Can Make It: Optimal Component Selection

Lars Gesellensetter\textsuperscript{1}  Sabine Glesner\textsuperscript{1}

\textsuperscript{1} Institute for Software Engineering and Theoretical Computer Science, Technical University of Berlin, Germany
Email: \{lgeselle|glesner\}@cs.tu-berlin.de
URL: http://pes.cs.tu-berlin.de/

Abstract
In Component-based Software Engineering (CBSE), the construction of cost-optimal component systems is a nontrivial task. It requires not only to optimally select components and their adaptors but also to take their interplay into account. In this paper, by employing methods from the area of compiler construction and especially optimizing code generation, we present a unified approach to the construction of component systems, which allows us to first select an optimal set of components and adaptors and afterwards to create a working system by providing the necessary glue code. With our two case studies, we demonstrate that our approach is efficient and generally applicable in practical scenarios.

\textbf{Key words:} component-based software engineering, component selection, adaptor code generation, term rewriting, cost functions

1 Introduction

Component-based software engineering (CBSE) has emerged as a method of choice to cope with more and more complex software systems and constantly increasing application fields. CBSE requires software functionality to be encapsulated into software components which in turn can be reused in further systems. In this paper, we address the problem of generating component systems from specifications of individual components. In general, there may be different alternative components that can be selected, each coming at their own cost. We aim at a selection method that guarantees the optimality of the generated component system. Especially in the presence of huge component repositories, the best solution is not trivial to obtain.

We require our solution to fulfill the following requirements: The components selected for the target system shall be optimal. In the classical case, the cost measure mirrors the speed of the target component system. In addition, we want to be able to deal with arbitrary cost functions, in particular
with those that are relevant in the area of embedded systems, e.g. code size or power consumption. Moreover, we require our method to be able to build component systems from existing components without modifying the components themselves, i.e. by regarding the components as black boxes. Merely the adaptor code shall be generated to connect the selected components into a working target system.

Our solution approach transforms the problem of constructing component systems from existing components to the problem of code generation in compiler backends, which is typically tackled with term rewriting methods. During code generation, we have to find an optimal sequence of machine instructions that not only accomplish the computations specified in the intermediate compiler representation but also adhere to the constraints imposed by the target processor, e.g. limited number of registers or functional units. This code generation problem in compilers can be solved by representing the program as a tree that can be reduced in bottom-up order while simultaneously generating semantically equivalent machine instructions for the reduced intermediate operations. Here in the context of component-based software engineering, we need to select components and adaptor code such that the generated component system offers the required services at optimal cost. For this purpose, we specify the desired behavior of the target component system as a term. Rewrite rules map the required services in the specification to concrete components. In the optimal solution, all required services are bound to components without conflicts at minimal cost. Moreover, the components are connected by appropriate adaptor code, if necessary.

Our work is a major contribution to the field of component-based software engineering as it improves reusability of components and automation in the construction process of component systems. Moreover, our method is generally applicable, not only for software components but also for hardware components or in the area of hardware/software-codesign as well. In all these cases, components are described by the functionality that they offer and need to be selected depending on their individual cost and on the context of the overall system in which they are employed. Adaptor code (which can be both software or hardware) may be needed for the integration of the selected (software or hardware) components.

In this paper, we discuss two case studies. The first considers the situation when designing the electronic system of a car. Typically, several subsystems are needed, let us assume, we need e.g. a navigation system, a CD player and a car theft protection system. Each of these subsystems can be realized in different ways, each coming at its own cost. For example, a car theft protection system may work by constantly sending the current position of the car. In this case, a GPS signal sensor is necessary. The cost for this design decision can be compensated if the GPS sensor is used by other subsystems as well, e.g. by the navigation system. In general, the overall cost of the car system depends not only on the costs of the individual components but also
on their interplay. We discuss the car scenario in more detail in Section 7. Our second case study considers a chatterbot system realizing a virtual partner for talks as an entertainment application for mobile phones [8]. This system offers a range of services, from a user interface to a dialog system to a visualization component. For each service, different realization possibilities of different costs exist. In Section 7, we show how our construction method can automatically select components such that the resulting component system offers all required services at optimal cost.

This paper is structured as follows: In Section 2, we introduce our notion of a component. Based on this definition, Section 3 gives an overview of our approach of component selection and integration. Section 4 deals with preliminaries concerning term rewriting and optimal code generation, which we need to explain our concept in detail. Section 5 gives a comprehensive presentation of our method. In Section 6, we describe the implementation of our approach, and Section 7 presents the introduced case studies in more detail. After discussing related work in Section 8, we conclude in Section 9.

2 Component Model

We describe system and component behavior by the means of services. A service encapsulates a certain functionality and is formally described by the interface of the methods that implement it. Each method is identified by its name and a list of named arguments. At the moment, we do not consider typing issues and consider the arguments simply as strings. Furthermore, we assume a black-box component model and describe components by four characteristics. Components are specified by the services that they provide (provided services) as well as by the services that they require in order to work properly (required services). Moreover, each component may pose architectural constraints (the required underlying platform in case of software components, communication standards, etc.). Furthermore, certain component properties (utilized for the cost measure) are known about the components, e.g. their code size, their energy consumption or their price. Figure 1 shows a component: A and B are the provided services and C is the required service. ArchStyle denotes the architectural constraints.

We assume the following situation: Given a repository of components and a specification of the component system that we want to construct, we need
to choose components and to connect them, if necessary, by suitable adaptor code such that the target component system fulfills the specification.

For this, we need a specification, if and how services can be mapped to each other. As illustration, this information can be thought of as (but not necessarily implemented as) a huge table, as depicted in Table 1. In the table, we see such a mapping, specified for a system of n components providing m services. The m services can be subdivided into k provided services (denoted as $P_i$) and l required services (denoted as $R_j$). Talking about adaptability, we always think of a provided service that is mapped to a required service. Such a pair of services can be incompatible, adaptable or identical. Adaptable means that the provided service’s methods can be mapped onto the required one’s. The mapping does not need to be one-to-one, and method names as well as arguments can be transformed. As we can see in the table, $P_4$ and $R_2$ as well as $R_l$ are identical and hence can directly interoperate. Service $P_k$ can be adapted to service $R_1$, and the same hold for $P_{k-1}$ and $R_2$, and for $P_1$ and $R_l$, respectively. In all other occasions, adaptation is not possible, the services are incompatible, indicated by a dash in the corresponding table cell. From the specification, we can see that the required service $R_2$ can be provided directly by service $P_4$ as well as via adaptation by service $P_{k-1}$. Depending on the costs of each alternative, one has to decide which choice to make.

In our implementation and in the case studies described in this paper, the mapping is efficiently given by a functional specification, from which the adaptor code can be generated by generic rules. The advantage is that these rules have to be specified only once for each considered architecture.

In case of several possibilities, we are interested in the optimum. We describe a component system (that we aim to construct) by the desired services. Additionally, we can define architectural constraints. Then, components have to be chosen according to their provided services. In the process of system

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Components/Services & $C_1$ & $C_2$ & \cdots & $C_n$ \\
\hline
Services & $R_1$ & $R_2$ & $R_3$ & \cdots & $R_{l-1}$ & $R_l$ \\
\hline
\hline
$C_1$ & $P_1$ & $-$ & $-$ & $-$ & $-$ & $M_{1l}$ \\
$P_2$ & $-$ & $-$ & $-$ & \cdots & $-$ & $-$ \\
$P_3$ & $-$ & $-$ & $-$ & $-$ & $-$ & $-$ \\
\hline
$C_2$ & $P_4$ & $-$ & id & \cdots & $-$ & id \\
\hline
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\hline
$C_n$ & $P_{k-1}$ & $-$ & $M_{k-12}$ & \cdots & $-$ & $-$ \\
$P_k$ & $M_{kl}$ & $-$ & \cdots & \cdots & $-$ & $-$ \\
\hline
\end{tabular}
\caption{Specification of the Mapping}
\end{table}
construction, the selection of a component might entail the selection of another one, since not all required services may be available. It has to be affirmed that the architectural constraints of the selected components fit to each other and to the initially given system constraints. Additionally to the given constraints, new constraints may evolve dynamically during the selection process. To select the optimal choice of components, we need a notion of optimality. To this end, a cost function has to be given. Usually, it assigns costs to the components by taking their properties into account. Moreover, the cost function assigns costs to the adaptor code that connects components.

Different services may be functionally equivalent but nonetheless might their interfaces be not directly compatible. In this case, an adaptor specification states how the interface of one service can be mapped to the interface of another one. If an adaptor maps the interface of a service $A$ to the interface of service $A'$, it can be regarded as a component which requires service $A$ and provides service $A'$. From the functional adaptor specification, an instantiation of the adaptor can be generated. This concept of adaptors allows us to make two components interoperable without intrusion, i.e. without changing the components, and, hence, fits well into our model of black-box components.

For the description of target component system to be constructed and the repository of available components, we specify the following information:

1. the description of the services in the target component system, specified by their interface descriptions,
2. the component specifications, which consist of the provided and required services as well as the architectural constraints and the respective properties defining the cost measure,
3. the adaptor specifications, which specify for each pair of components if and by which adaptor code their services (i.e. their interfaces) can be mapped onto another.

As an example, consider Figure 2: The system has to provide services $A$ and $B$. Depending on the given repository, we might have components at hand, which provide service $A$ and $B$, respectively. If their constraints match each other and also the initial constraints, we have one solution as depicted in Figure 2(i). However, we might choose a single component that provides
A and B, but which requires another service C. Then it can be the case that we only have a component available that provides service C'. In case we have an adaptor specification for the mapping from C' to C, we arrive finally at the solution depicted in Figure 2(ii). It depends on the cost function which solution is preferable. If e.g. the left component in Figure 2(ii) is cheaper than both components in Figure 2(i) together, the second choice can be better. However, the cost of having to select an additional component for service C might even out this benefit or even lead to a suboptimal solution. In the following section, we give an overview about our approach to the construction of cost-optimal component systems.

3 Constructing Optimal Component Systems by Term Rewriting

In this section, we give an overview about our approach, especially about the idea of employing methods from the area of compiler construction. We start with the following concrete problem: We assume that the desired system behavior is given in form of the wanted services together with a specification of the available services, components and adaptors together with their cost function and the architectural constraints. Given this description, we want to find an optimal set of components that fulfill the requirements and achieve interoperability (by regarding the constraints and generating adaptors).

In general, this problem is NP-complete \[10,11\] so that we cannot hope for an algorithm that always solves each instance of this problem efficiently, i.e. in polynomial time. Nevertheless, we can investigate heuristic strategies that may help to find optimal solutions in many practical cases. For this purpose, we map our problem to the problem of optimal code selection in compiler backends. This is especially useful as in both cases, we have a description of the system behavior from which we need to generate the target system. In case of component systems, the description specifies the desired services. In case of compiler backends, it expresses the desired functionality as intermediate compiler representation. In both cases, we have parts of the system which achieve a certain functionality (components or machine instructions of the target platform, resp.). In both cases, the mapping is rather 1 : n than 1 : 1 since a component can provide many services, and likewise a machine instruction can perform a composed functionality\(^2\). Finally, in both cases we are interested in the optimal solution. Hence it is a promising approach to transfer solutions from the code generation problem to component selection.

For optimal code generation, one standard technique is term rewriting. It finds provably the optimal solution. In term rewriting, we start with an intial

\(^2\) Consider complex load operations, which perform addition and multiplication to determine addresses. Hence, a load with address calculation can be mapped to multiple simple commands or to the complex load instruction. Another example is multiplication by 2 which can be realized by a multiplication operation or by the much cheaper shift-left operation.
term and try to rewrite it in bottom-up order to a goal term, by using a set of given rewrite rules. Whenever a rewrite rule is applied, semantically equivalent machine code is emitted. The total sequence of emitted machine code makes up the target machine program. Each rule is adherent with a cost. In code generation, bottom-up term rewriting is used, to both restrict the search space and ensure the correct ordering of the generated instructions (cp. [9]; [1] proves optimal code generation for expression trees with shared subexpressions as NP-complete). Since there are usually many ways to rewrite a term, we deal with a classical search problem. Heuristic search is a valuable mean to cope with the complexity of the search space. The A*-search guarantees to explore a minimal part of the search space, while still finding the optimal solution, in the sense that no other algorithm searching for the optimum is guaranteed to expand fewer nodes than A*, cf. [6,3]

To model the problem of component selection as term rewriting problem, we state the desired system behavior as a term. The rewrite rules correspond to selection of a component (together with its provided and required services) or of an adaptor. The cost function for each rule expresses the cost of the selected component. Hence, we denote a system specification in the following way:

\[
\text{system}_{\text{constraints}}(\triangle^A, \triangle^B)
\]

This term represents a system, which should provide services A and B under the given architectural constraints. The \(\triangle\) denotes that the given service is still unbound. During the rewrite process, each service will be bound to certain components, and further services might be introduced, if they are required by a selected component. At the end, we arrive at a final state, namely a term in which all components are bound. The collected information during the rewrite process tells us which components to select and how to achieve interoperability (by generating adaptors).

Reconsidering our previous example, we can specify the desired system behavior by the following term:

\[
\text{carsystem}(\triangle^{\text{anti-theftprotection}}, \triangle^{\text{stereo}}, \triangle^{\text{navigationssystem}})
\]

One rewrite sequence could be (abbreviating the services for convenience):

\[
\begin{align*}
\text{carsystem}(\triangle^{\text{ATP}}, \triangle^{\text{stereo}}, \triangle^{\text{NavSys}}) & \rightarrow \text{carsystem}(\text{GPSLoc}^{\text{ATP}}, \triangle^{\text{stereo}}, \triangle^{\text{NavSys}}, \triangle^{\text{GPS-sensor}}) \\
& \rightarrow \text{carsystem}(\text{GPSLoc}^{\text{ATP}}, \text{CDman}^{100\text{stereo}}, \text{NAV}^{200\text{NavSys}}, \triangle^{\text{GPS-sensor}}) \\
& \rightarrow \text{carsystem}(\text{GPSLoc}^{\text{ATP}}, \text{CDman}^{100\text{stereo}}, \text{NAV}^{200\text{NavSys}}, \text{GPS}^{10\text{GPS-sensor}})
\end{align*}
\]

Note the introduction of a new service in the first rewrite step. For the first rewrite step, the applied rule is shown in Figure 3. We see that the rule is applicable, since we have an unbound service of type \text{ATP} \footnote{In this example, there are no further architectural constraints.} . As a result of
Fig. 3. Example of a rule for component selection

the rule application, the service \( ATP \) is bound to \( GPSLoc \), and \( \Delta^{GPS-sensor} \) is added as new required service, since it appears neither as bound nor as required service in the current system description. The remaining rewrite steps are performed similarly by corresponding rules.

After the general presentation of our concept and some motivating examples, we are now ready to describe our approach in detail. To this end, we introduce some preliminaries in Section 4, which we need for our concept, namely term rewriting and \( A^* \) search. Having established these prerequisites, Section 5 presents our approach.

4 Term Rewriting and \( A^* \)

In our approach, we use the idea of coupling term rewriting with \( A^* \) as proposed by [9]. This guarantees an efficient search. We give a short overview of term rewriting with respect to cost functions, and introduce then the concept of \( A^* \) search. Finally, we describe the coupling between both and discuss complexity issues.

4.1 Term Rewriting

Term rewriting [2] is a standard technique for code generation. Terms are defined inductively in the usual way: We start with a finite set of function symbols \( \mathcal{F} \) which is called signature. Each function symbol has an arity given by the function \( r : \mathcal{F} \rightarrow \mathbb{N} \). Function symbols with arity 0 are also called constant symbols. We add a set of variables \( \mathcal{V} \) \((\mathcal{F} \cap \mathcal{V} = \emptyset)\). The set of terms, denoted as \( T(\mathcal{F}, \mathcal{V}) \), is defined in the usual inductive way. For a term \( t \), \( Var(t) \) denotes its variables. If \( Var(t) = \emptyset \), \( t \) is called a ground term.

**Definition 4.1 (Costed term rewrite system)** A costed term rewrite system (CTRS) is defined as a triple \((\mathcal{F}, \mathcal{V}, R, C)\) with

- \( \mathcal{F} \), a non-empty signature
- \( \mathcal{V} \), a finite set of variables
- \( R \), a non-empty, finite subset of \( T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V}) \)
- \( C \in R \rightarrow \mathbb{R}^+ \), a cost function
We enrich the notion of a term by allowing arbitrary arity for certain function symbols (to model lists adequately) and attach a type and attributes to the function symbols. Besides, we extend the rules by preconditions, which can refer to any of this information. Note that these extensions do not introduce additional complexity, since they can be straightforwardly expressed in classic term notion (model attributes as children, model n-ary function symbols by introducing n new function symbols and modifying the rules accordingly). We extend and modify the given definitions by the following:

**Definition 4.2 (n-ary Terms)** We extend the domain of the arity function to \( \mathbb{N} \cup \{\ast\} \), \( \ast \) meaning that the arity is arbitrary but finite. The inductive definition of \( T(\mathcal{F}, \mathcal{V}) \) has to be modified accordingly.

**Definition 4.3 (Attributed Terms)** We introduce a function \( t : \mathcal{F} \to T \) that assigns a type to each function symbol. The attributes of a function symbol are determined by its type. To each function symbol in a concrete term, an attribute function is associated that returns the attributes of the term.

The type information allows us to write more generic rules. E.g. in the context of code generation, the function symbols for the arithmetic operation + and − would be both of type \( \text{BinOp} \). This allows us to write a generic rule dealing with binary operators.

For notation, we write the type of a function symbol superscribed and its attributes subscribed. If the type is irrelevant or clear from the context, we simply omit it, as we do with the attributes. A component \( \text{comp} \) of type \( \text{type} \) with attributes \( a_1, a_2 \) is hence denoted as follows:

\[
\text{comp}_{a_1=\text{val}_1, a_2=\text{val}_2}^{\text{type}}
\]

A rewrite rule specifies how terms can be rewritten, which preconditions have to be fulfilled and which costs arise. Additionally, it can be stated which code should be generated.

In general, there are many ways to rewrite a given term. To find the optimal rewrite sequence, all different possibilities have to be considered. The complexity of the search space has motivated the utilization of heuristic search methods, whereof the A* search is an instance.

### 4.2 A* Search

A search can be guided by a heuristic function to approximate the cost for the remaining transitions. The A* search ([6]) guarantees minimal exploration

---

\(^4\) Our notion of type is for now very basic and mainly denotes a special attribute, referring only to the given function symbol (and not the complete subterm).
Fig. 4. Search space and visited nodes for A* search

(w.r.t. the heuristic function) of the search space while finding an optimal solution, given that the heuristic function is **admissible**. **Admissible** means that the function always underapproximates the actual cost. If we take the zero function as heuristics, we end up with the same behavior as uninformed search. The more precise the heuristic function predicts the costs, the less expensive the search will be. The heuristics provides a valuable mean for dealing with the problem of combinatorial explosion.

A common and descriptive example for search is the problem of finding an optimal route from one location to another. The nodes correspond to the locations, and the edges represent connections, e.g. streets. The cost of a given edge is its distance. The heuristic function for a node defines its distance (here meant literally) to the goal node. Intuitively, we choose air-line distance\(^5\). As example, consider Figure 4. We would like to travel from A to Z, and our search tree consists of all possible paths starting from A and possibly leading to Z. At every node, we annotate the accumulated cost (the distance travelled so far) and the estimated remaining cost (how far is it). A* is a breadth-first search, i.e., we consider all possibilities but look at the cheapest first. For the given example, we first try B, since 12+40 = 52 is the cheapest choice. However, having reached B, we have a cheaper alternative to E. Hence backtracking occurs and we continue at D. The remaining search leads straightforwardly via G to the goal Z, encountering an actual distance of 60 instead of the initially assumed 50. The shaded area in Figure 4 depicts the visited section of the search space.

4.3 **Coupling Term Rewriting with A***

It is a general search problem to find the optimal solution for a given term. The search space contains all terms, the transitions are given by the rules. If we take A* search with an admissible heuristic function, it is guaranteed that we find the optimal solution in minimal time with respect to that heuristics. This application of A* to term rewriting in the context of code generation in

\(^5\) Trivially, this heuristics always underestimates the actual cost.
compilers has been initially proposed by [9].

4.4 Complexity

As mentioned before, selecting a subset of components for a given set of services is NP-complete (this problem is in NP and is reducible to SAT), even without the constraint to find an optimal solution. Considering term rewriting, in general it is undecidable whether a given term is reachable (corresponding to finding a solution), even termination alone is undecidable\(^6\). Restricting the rewriting to ground terms (i.e. forbidding variables in the rules), reachability of a goal term is decidable. Least-cost instruction selection on DAGs is NP-complete [1]. While ground term rewriting has to deal with the problem of combinatorial explosion, it nevertheless yields a constructive algorithm for finding a solution, and by restricting the class of allowed rules and appropriate reduction of the search space, the problem becomes feasible for practical scenarios.

5 Composing the System: The Algorithm

The composition of the system is realized in two steps: First the component selection is accomplished (which includes selection of adaptors). In the second step, the required adaptor code is generated. The system specification is represented as a term, and component selection as well as adaptor code generation are performed by term rewriting. The rules are partly general (especially for the adaptor code generation), and partly specific for the concrete system (especially for the component selection). However, the specific rules are fully automatically generated from the system specification.

5.1 Component Selection

A system description consists of the descriptions of the components, of the mapping from provided services to required services, and of the desired functionality. Additionally, a cost measure has to be specified.

Generally, in describing a system, the relevant services have to be identified. By identifying we mean that the functionalities of the components are grouped into self-contained, meaningful aggregates referred to as services.

The description of the system behavior is mapped to a term, the arguments of which specify the required services. In the rewrite process, as components are selected, the corresponding provided services are bound accordingly, and new services might be introduced. Remember that selecting a component entails that all services in the system specification that are provided by this component are bound to it, given they were unbound before. To this end,

\(^6\) This holds even for quite simple systems with only one rule; shown by reducing it to the word problem.
component and adaptor descriptions are mapped to rewrite rules. The rules
generated from the adaptor specification replace a service with another one.
Therefore adaptors can be seen as a special case of a component. Hence in
the following, if we talk about component selection, we equally mean adaptor
selection. The generation of corresponding adaptor code is performed in the
next step.

As example, reconsider the following term from our running example:

\[\text{carsystem}(\Delta^{\text{ATP}}, \Delta^{\text{stereo}}, \Delta^{\text{NavSys}})\]

This term describes a system specification, which states that the three ser-
vices anti-theft protection, stereo, and navigation system are required. Each
required service is modelled as argument of the root, whereas the type of the
argument denotes the required service, and the name states which component
(if any) is bound to the service. The special name $\Delta$ means that the service
is still to be bound.

The selection of components is achieved by the application of rewrite rules,
which are automatically generated from the component specification. A rule
for a component $C$ with required services $\text{req}(C)$, provided services $\text{prov}(C)$,
constraints $\text{constr}(C)$ and properties $\text{props}(C)$ is depicted in Figure 5 (with
cost being the given cost function, built upon the properties). A rule is consti-
tuted by a condition, a matching expression, a replace expression, an assert,
and a cost function. For a rule to be applicable, its condition has to be fulfilled,
and its match expression has to match the considered term. For component
selection, the condition states that the constraints of the component and the
current system constraints are consistent. The match expression states that
there should be an unbound service, which is provided by the considered com-
ponent. If an applicable rule is selected and applied, the matched expression is
replaced with the replace expression. In case of component selection, the list of
services which constitute the system behavior is modified. Unbound services
which are provided by the considered component are bound to this compo-
nent. Services which are required by the component are added as unbound

```
rule "general rule for component selection" {
  cond: constr(System) ∧ constr(C) satisfiable
  match: system[[services]] | ∃ S ∈ services : S ∈ prov(C)
  replace: system[[services']], where services' =
    {CSprov | ∆Sprov ∈ services ∧ Sprov ∈ prov(C)}
    ∪{CSreq | Sreq /∈ services ∧ Sreq ∈ req(C)}
    ∪{C̄S| S ∈ services | (C̄ ≠ ∆) ∨ (S /∈ prov(C))}
  assert: constr(System) ∧ constr(C) satisfiable
  cost: cost(props(C))
}
```

Fig. 5. Generic rule for component selection
services, unless they are already bound in the previous system configuration. And finally, services which neither are unbound nor are in the list of provided services of the considered component are simply kept. This yields a new list of services (compare the replace-expression in Figure 5). Then, after the application of the rule, we can assert that the constraints of the component and the current system constraints are consistent.

Example 5.1 If we recall the system in fig. 2(i), this solution might be reached by the following steps: select component C1, select component C2. This is represented by the following rewrite sequence:

\[
\text{system}_{\text{arch}}=\text{i}386(\Delta^A, \Delta^B) \\
\rightarrow \text{system}_{\text{arch}}=\text{i}386(C1^A, \Delta^B) \\
\rightarrow \text{system}_{\text{arch}}=\text{i}386(C1^A, C2^B)
\]

On the other hand, the solution in Figure 2(ii) could be established by first selection component C3, and then choosing component C4 (after choosing the adaptor A1), which is expressed as the sequence:

\[
\text{system}_{\text{arch}}=\text{i}386(\Delta^A, \Delta^B) \\
\rightarrow \text{system}_{\text{arch}}=\text{i}386(C3^A, C3^B, \Delta^C) \\
\rightarrow \text{system}_{\text{arch}}=\text{i}386(C3^A, C3^B, A1^C, \Delta^C') \\
\rightarrow \text{system}_{\text{arch}}=\text{i}386(C3^A, C3^B, A1^C, C4^C')
\]

Note that the second rewrite step (changing C to C’) corresponds to the selection of adaptor A1, and demonstrates that adaptors are dealt with as components.

Since all rewrite steps come at a certain cost, first the best options (i.e. the ones with the lowest costs) should be considered. If we specify a heuristics, we can optimize the search. The A* search requires a heuristic function, which estimates the cost from a given term to the goal term. Note that the function must always underestimate the actual cost to guarantee the optimality of the A* search. If we know that each service comes at least at a cost of \(c_{\min}\), a simple heuristics would be the following: From a given term with \(n\) unbound services, the expected cost is at least \(n \times c_{\min}\). Although this heuristics seems quite simple, it is very effective in reducing the number of visited nodes.

Figure 6 specifies our algorithm for component selection. Our search space does not only contain the mere terms but instead terms with information about the current state of the rewrite process annotated, called \(x\text{terms}\). For each \(x\text{term}\ \(t\), we have the following:

- \(\text{term}(t)\): the current term
- \(\text{constr}(t)\): the currently posed constraints
- \(\text{seq}(t)\): the rewrite sequence encountered so far, each step consists of the application of a rule \(r\) at a position \(p\)
**func** selectComponentsAndAdaptors: (t:XTerm) → XTerm

**func** filter (terms:XTerm list, newterms:XTerm list) → XTerm list

(* eliminate redundant xterms in newterms (w.r.t. terms)
  redundancy is meant modulo equivalence of rewriting sequences *)

**func** result (term:XTerm) → ResultType

(* returns the desired result. This can be the list of generated code,
  or (in the case of component selection) the final term that
  provides all information about the component selection.*)

**func** getNextTerms (t:XTerm) → XTerm list

newTerms=[];
for all rules r=(cond,match,replace,assert,cost) do
  for all positions p of term(t) do
    if "match matches term(t) at position p and
       (constr(t) ∧ cond) is satisfiable " then
      t'=t;
      "modify term(t') according to replace, add (p,r) to Seq(t'),
       set estcost(t')=estcost(t)−heurCost(t)+cost+heurCost(t'),
       add cond to constr(t')"
      newTerms=newTerms ∪ [t'];
    endif
  endfor
endfor

return newTerms;
endfunc

**var** terms: XTerm list;
var selected=t;
while (terms≠[]) and (not (goalTerm(selected))) do
  newTerms=filter ( getNextTerms(selected) );
  terms=(terms \ [selected] ) ∪ newTerms;
  if (terms ≠ []) then
    selected=selectMinimum terms;
  endif
endwhile
if (goalTerm(selected)) then (* we found the optimal solution *)
  return result ( selected );
else return failure ;
endif
endfunc

Fig. 6. Algorithm for component selection
• **estcost(t):** the estimated total cost of the term (contains actually encountered and estimated remaining cost)

Additionally, we have a heuristic function \( heur : xterm \rightarrow \mathbb{R}^+ \), which estimates the remaining costs for a node.

The algorithm works as follows: We keep a list of currently active xterms. In each iteration step, we select the minimum (with respect to the cost function) of this list and replace it with all successors which could be generated by application of a rewrite rule. We finish if we reach a goal term, in which case we report success, or if no terms are remaining to be examined, which means failure. If we end up with a goal term, the selection of the minimum in the loop guarantees that we found the optimal solution.

The determination of the successors of a xterm \( t \) considers all rules and all positions of \( t \) to find possible rewrite steps. If a rule \( r \) matches at a given position \( p \), and if the constraints of the rule meet the current system constraints, according to the replace-expression a new term \( t' \) is generated. Additionally, we extend its rewrite sequence, its constraints, and its estimated cost. Since we only remember the estimated cost, a little calculation has to be done to this end: The previously encountered costs of \( t \) result to \( estcost(t) - heur(t) \). Therefore at node \( t' \), to get the actually cost we simply have to add the cost of the rule application. By finally adding \( heur(t') \), we arrive at the estimated cost for \( t' \), as we see in the line denoted with \([\ast]\).

Having established the algorithm for component selection, in the following we sketch how the generation of adaptor code works.

### 5.2 Adaptor Code Generation

After we have selected components and also adaptors (as a special case of the former) in the first step, in the second step we generate code for the chosen adaptors. For this, we need the adaptor specification which states how the interfaces of two services can be mapped to each other (cp. Table 1). The description covers binary directed mappings. The adaptor specification can be rewritten by general rules to the desired target platform.

For the specification of an adaptor, we need a notion of how communication between components is modelled. We assume that the components communicate by message exchange. Each message has a name (or a type) and a number of named arguments (i.e. a list of attribute-value pairs). If two services are functionally equivalent, but not per se interoperable, the cause can be one of the following:

- related messages have different names
- related messages have different names for their arguments
- \( n \) messages of one service can be mapped onto one message of the other one
- the domains of related arguments are different

Under these circumstances, a mapping between the corresponding services
can be specified. This mapping can be straightforwardly transformed to an implementation. Since for now, an adaptor specification consists basically of this mapping, rules for generating the adaptor code can be easily written. One simply has a fixed frame for each adaptor and generates if-then-statements for each mapping. For now the adaptor specification has to be developed manually. However, one can think of generating this information automatically as well, by considering the given components and figuring out how they can interoperate.

6 Implementation

We have implemented our approach for the construction of component systems based on term rewriting in ML. ML is particularly suited for this purpose because of its higher-order power. In our implementation, a higher-order function realizes the generator generator. Given a concrete specification of the available components, it outputs a function that, given the desired component system specification as input, outputs a set of components and adaptors that implement the desired target system.

Terms are represented via a generic datatype. The term rewriter starts with the input term, computes all possible successors and puts them into a candidate list. During the following, the least-expensive (w.r.t. the previous cost and the expected cost) candidate is removed from the list and replaced by its successors. This process continues until a goal term is reached.

The system specification is given by the following datatypes:

```
datatype Message = Msg of Name * (Argument list)
datatype System = SystemSpec of Service list

datatype Service = ServiceSpec of Name * Message list

datatype Component = ComponentSpec of Name
       * Constraint list
       * ProvidedService list
       * RequiredService list
       * Property list

datatype Adaptor = AdaptorSpec of Name
       * ProvidedService
       * RequiredService
       * MsgMapping list
```

The mapping specifies how an incoming message is transformed to the message to be sent. In the transformation, arguments can be transformed, renamed, the message can be renamed, and an internal state can be updated.

The term rewriting rules have the following origin:

- The rules for component selection are generated from the component specifications. For each component, a rule is generated and corresponds to the selection of that component (as described in detail in the previous section).
The rules for adaptor selection are also generated from the specification, as a special case of component selection.

The rules for adaptor code generation depend only on the target platform and have to be specified manually. However, having them specified once allows for automatic generation of adaptor code for a given specification.

The role of the heuristic function must not be neglected. As an example, let us consider a first toy example we have modeled, consisting of 4 services and 7 components. Even for such a small setting, the search space contains 174 nodes. To find the optimum, 25 nodes were visited with uninformed search. If our earlier explained heuristics is used (minimal cost for a component selection presumed), only 10 nodes were visited, while still yielding the same result.

### 7 Case Studies

In this section, we discuss our two case studies. We first revisit the example from the introduction for an automotive system. Then, we present a case study in which we have modelled a software component system in the EJB (Enterprise Java Beans) framework [4]. Taking a repository of components for granted, a running system has been created (inclusive adaptors) from the specification.

#### 7.1 Automotive Infotainment Unit

Let us reconsider the first example from the introduction. We want to select components in an automotive context. Our system should contain the following functionality: navigation system, cd player, car-theft protection. We assume that we have certain components at our disposal and are now to select the best composition. Table 2 gives an overview of the components. For each component, the offered as well as the required services are denoted. Additionally, the associated cost (e.g. the price) is shown.

Given our requirements, the optimal solution is the component selection GPS Locator, CDman 100, NAV-200, GPS 10 with a cost of 230 (note that the selection of either of NAV-200 and GPS Locator forces to select a component which provides the service GPS sensor). However, if we omit the require-

<table>
<thead>
<tr>
<th>component</th>
<th>offered services</th>
<th>required services</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS Locator</td>
<td>anti-theft protection</td>
<td>GPS sensor</td>
<td>50</td>
</tr>
<tr>
<td>CDman 100</td>
<td>cd-player</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>NAV-200</td>
<td>navigation</td>
<td>GPS sensor</td>
<td>80</td>
</tr>
<tr>
<td>NAV-300</td>
<td>navigation</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>NAV-1000</td>
<td>navigation, cd-player</td>
<td></td>
<td>160</td>
</tr>
<tr>
<td>GPS 10</td>
<td>GPS sensor</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

Table 2
Automotive Infotainment Unit: Component repository
ment for the anti-theft protection, the integrated cd-player/navigation system *(NAV-1000)* will be the best solution (cost: 160).

If we model the initially given example, our compacted search space consists of 39 nodes. To find the optimum, 23 (20) nodes are visited (with heuristics presuming minimal component cost)\(^7\).

### 7.2 Entertainment Software for Mobile Devices

As second case study, we have modelled an application intended for entertainment on handheld devices. The application should act as a virtual conversational partner for the user, as a so-called *chatterbot*\(^8\). We used the Enterprise Java Beans (EJB) component framework as target platform.

The services which can be identified are: DialogSystem, Visualization, UserInterface. The available components are listed in Table 3, each component corresponds to a Java Bean. As cost measure, the size of the binaries (in kilobyte) was chosen. The constraints in this context describe the display type, namely whether it is black-and-white or color, and whether it is text-based or graphical. Mutually exclusive constraints have to be modelled as one parameter, e.g. black-and-white and color are mapped to the boolean value *color*. The specification for two components are given below.

<table>
<thead>
<tr>
<th>component</th>
<th>proviced services</th>
<th>required services</th>
<th>constraints</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>text input</td>
<td>UserInterface</td>
<td></td>
<td>b/w, text</td>
<td>10</td>
</tr>
<tr>
<td>text input color</td>
<td>UserInterface</td>
<td></td>
<td>color, text</td>
<td>13</td>
</tr>
<tr>
<td>graphical input</td>
<td>UserInterface</td>
<td></td>
<td>color, graphic</td>
<td>100</td>
</tr>
<tr>
<td>eliza</td>
<td>DialogSystem</td>
<td></td>
<td>UserInterface, Visualization</td>
<td>80</td>
</tr>
<tr>
<td>smiley color</td>
<td>Visualization</td>
<td></td>
<td>color, text</td>
<td>20</td>
</tr>
<tr>
<td>face color</td>
<td>Visualization</td>
<td></td>
<td>color, graphic</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 3

Entertainment Software: Component repository

The components are automatically transformed into rewrite rules. As example, we regard the rewrite rule for the selection of the component *text input color*, cf. Figure 7. As we see, this component can only be selected if the

\(^7\) Without search space compaction, out of 163 nodes 48 (35) were visited (with heuristics).

\(^8\) The setting is a simplification of a system presented in \[8\].
already chosen architectural constraints do not contradict to the constraints of the component. After selection, the constraints of the system are extended by the constraints of the component.

The process of component selection can start with no constraints at all. In this case, constraints are dynamically selected during the rewrite process. On the other hand, we can preselect certain constraints. In the given example, if we start with no constraints, the optimal solution is the following: Select components eliza, text input color, smiley color, at the cost of 113. If we start with the constraint graphic=yes, we find this optimal solution: eliza, graphical input, face color, leading to a cost of 330.

For brevity, the description of adaptor specification has been ommitted. However, in the case study, the interfaces of the different components did not match to each other, and interoperability was only achieved by adaptor specification. Starting with the specification of the system (services, components, adaptors) and with the given Java Beans for the components, components were selected and integrated by adaptor code generation in a complete automatical process, leading to a deployed application.

For this case study, the compacted search space contained 18 nodes at maximum depth 5. For the optimal solution, 12 (11) nodes were visited (with heuristic function)\(^9\). The search space is depicted in Figure 8.

With these two case studies, we have demonstrated that our term-rewriting based approach to the construction of component systems is efficient and

\(^9\) The full search space contained 45 nodes, 16 (15) nodes being visited (with heuristics).
suites for practical applications.

8 Related Work

Our approach uses ideas from the algorithm proposed in [9] for optimal code generation in compilers. However, their approach puts strong constraints on the rewrite sequence to reduce the search space. In a precomputation step, the given term is annotated with possible rewrite sequences. As a result, the algorithm cannot cope properly with dynamically added subterms. We have modified their method by allowing not only bottom-up term rewriting but also more general rewrite orders.

An analysis of the complexity of component selection is given in [10,11]. The authors establish that component selection is NP-complete. [7] considers as well the problem of component selection and models the problem as retrieval problem.

Concerning the problem of the generation of adaptors, [14] makes an important contribution. They model protocols with finite state automata and provide a formal definition of properties like dead-lock-freeness and liveness. Starting from a functional description of an adaptor, they show how to generate the corresponding adaptor code.


Most related work deals with either component selection or adaptor code generation. The contribution of our approach is the unified view of both processes.

9 Conclusion

We have presented a framework for the construction of optimal component systems based on term rewriting strategies. By taking these techniques from compiler construction, especially optimizing code generation, we have been able to develop an algorithm that finds a cost-optimal component system in minimal (wrt. the applied heuristics) time. The importance of the heuristics is very important as it directly determines the efficiency of the construction algorithm. In our case studies, we have demonstrated that a good heuristics can reduce the search space drastically.

Our method is very general as it does not put many requirements on the kinds of components and adaptors that can be selected and composed into a working system. In future work, we want to apply our construction method to hardware and mixed hardware/software systems as well. The constraints that these kinds of components need to fulfill can then be expressed in the architectural constraints of our formalism. Moreover, we want to extend our method such that not only two components but also larger numbers of components
can be connected by suitable adaptors. And finally, we want to formally verify that the constructed component systems behave correctly as described in their specification. For this purpose, we also want to apply methods from compiler construction, especially verification methods [5] that can be used to show that transformation algorithms as well as their implementations are correct.

References


Local Module Checking for CTL Specifications

Samik Basu
Department of Computer Science
Iowa State University
Ames, USA

Partha S Roop, Roopak Sinha
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand

Abstract
Model checking is a well known technique for the verification of finite state models using temporal logic specification. While model checking is suitable for transformational systems (also called closed systems), it is unsuitable for open systems (also known as reactive systems) where the nondeterminism in the environment must be considered during verification. Module checking is an approach for the verification of open systems which have both closed (internal) and open (environment or external) states. It has been demonstrated in [10] that the complexity of module checking branching time logic CTL is EXPTIME-complete. The approach to module checking is global and the method tries to establish that the property in question holds over all possible environments.

This paper develops a local approach to CTL module checking using tableau rules. The proposed approach tries to determine a single environment under which the negation of the property is satisfied over the given module. Such a strategy, thus, leads to a local approach to module checking where we only explore states that are relevant to proving that the negation of the property can be satisfied over the given module using an appropriate witness (environment) that the algorithm also generates. While the worst case complexity of our algorithm is identical to the earlier complexity, we demonstrate that practical implementation of the proposed approach is feasible and yields much better results than the global approach.

Key words: module checking, open systems, tableau based verification

1 Email: sbasu@cs.iastate.edu
2 Email: p.roop@auckland.ac.nz
3 Email: rsin077@ec.auckland.ac.nz

This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
1 Introduction

Reactive systems [7,13] are open systems that continuously interact with their environment while executing a non-terminating control program. Examples include operating systems, communication protocols, missile and avionics systems and controllers for nuclear plants and simple home appliances such as microwaves, DVD players and washing machines. Such applications require careful analysis, design and validation techniques as they can often be safety critical. Hence, formal techniques have often been used in the design, development and validation of these systems [16,11,5,6,3,12,15]. Formal techniques use precise syntax and semantics for defining specifications and models of systems so that rigorous verification of properties such as correctness, reliability and security is made possible.

With the advent and widespread use of embedded systems, which are ubiquitous reactive computing systems ranging from simple home appliances to very complex applications in avionics and defense, the need for formal methods in the design of reactive systems is growing. One very common approach to verification of reactive systems has been model checking. A model checker takes a formula in some temporal logic [14] as a desirable property and performs formal analysis over a finite-state model of a system (called a Kripke structure [3], a special class of finite state machines). The model checking process is essentially an automated reachability analysis task over the finite state model of the system. This task either terminates with a proof that the temporal property holds over the model or on failure generates a counter example.

The model checker, during the reachability analysis phase, assumes that all transitions out of any given state of a model is eventually enabled. This assumption is based on the fact that the model is considered to be closed i.e., all states of the system are purely internal and that it is the system that gets to choose which transition to take based on some internal computation. This approach to analysis, while being suitable for transformational systems which are closed, are unsuitable for reactive systems that are open.

An open system maintains an ongoing interaction with its environment. Hence, the state-space of such a system may be partitioned into a set of states that are open (also called external or environment states) and another set of states that are closed (also called internal or system states) [8,9]. An environment state reacts to events in the external environment of the open system and the environment is considered asynchronous and uncontrollable. A system state, on the other hand, takes no inputs from the external environment and the system automatically chooses one of the transitions based on some internal decision (such as say the value of a variable or the result returned by a function). Kupferman et al. have recently shown that model checking may not be enough for open systems due to the presence of environment states and when branching formulas are considered in the specification [9]. The proposed
technique, called module checking, takes the asynchronous environment into account while doing a proof for branching-time logics. [6,5] further discuss techniques devoted to issues concerning verification of open systems.

This paper illustrates the need for module checking reactive systems and proposes an alternate approach to module checking that has efficient implementation avenues compared to the original approach. We first illustrate the need for module checking followed by our approach to local module checking.

1.1 Motivating Example - The Coffee Brewer Verification Problem

Consider a simple coffee brewer model as depicted in the Figure 1. In this figure, environment states are shown using ellipses whereas system states are drawn as circles. Each transition out of any state is marked by a natural number starting with 1. The brewer serves either five or ten cups of coffee in either medium or strong flavor. A user can select the number of cups of coffee (using a switch as an input) and the strength of coffee (using another switch). The brewer is normally in the off state until the switched on. Hence, the initial state labeled by the proposition OFF is an environment state. Once the brewer is switched on, it enters a state labeled by the proposition CHOOSE. This is again an environment state. In this state, the user can select the number of cups of coffee and the strength of coffee. Depending on the selection (two switches lead to four different possibilities), the brewer enters any one of the following states: (five, medium), (five, strong), (ten, medium) or (ten, strong). Once the selections have been made and the corresponding state has been reached, the brew cycle switch has to be switched on to start the brewing. Hence all the above four states are also environment states. Once the brew cycle is set, the brewer makes a transition to a state labeled BREW. This state is a system state since no inputs from the environment is required to make progress. From the BREW state, the brewer makes an automatic transition to the DONE state after a predetermined time period (which is the amount of time taken by the brewing process). The state labeled DONE is also a system state since the brewer takes one of two possible branches based on some internal condition. If any error is detected (say not enough coffee or no milk power), then a transition is made to an error state (labeled by ERROR). Alternatively, the brewer can reach a state labeled SERVE where the actual coffee selected is served.

CTL is a branching time temporal logic that has been shown to be quite efficient for model checking. Let us consider the following CTL property:

\[ \text{AGEF \((TEN \land \text{AF (SERVE \vee ERROR)})\)} \]

which demands that from any state one can possibly eventually select ten cups of coffee and once selected, ten cups will always be served (or an error encountered) in the future.

Note that a model checker will always return a true answer for this question. However, consider the following situation. Due to cost cutting in the work place where the brewer is installed, brewing ten cups of coffee at a time
is not allowed (this has been enforced through a circular and the ten cups switch is masked). Hence, no user will be allowed to make this selection from the CHOOSE state. Thus, it will not be possible to guarantee the selection of ten cups of coffee and hence the property fails to hold over the coffee brewer model. This property could also be violated if all users request five cups of coffee or tea (and no users request ten cups of any beverage). In this case, even though the machine is capable of dispensing ten cups of tea or coffee, the environment in which it operates effectively disables it from doing that.

This property illustrates that due to the presence of environment states, it may not be always possible to satisfy branching time temporal properties. As the environment of an open system is asynchronous and hence uncontrollable, the environment may never enable some desired transitions leading to failure of the property. This example thus motivates the module checking problem: how do we ensure that a given property holds over a known module (a model with environment and system states) assuming an uncertain environment. The basic idea in module checking is to prove that the property holds over all possible choices made in the environment states. In other words, the module checking problem is to decide if a CTL formula $\varphi$ holds over a module $M$ provided for all possible environments $E$ such that $M \times E$ satisfy $\varphi$. Here, $M \times E$ denotes the composition of the model and the environment running in parallel (to be formalized later).

This may be expressed as follows: module checking, denoted by $M \models_o \varphi$, therefore, amounts to deciding whether $\forall E. M \times E \models \varphi$; i.e.

$$M \models_o \varphi \iff \forall E. M \times E \models \varphi$$

(1)
It has been shown in Kupferman et al. [10] that the module checking problem for the temporal logic CTL is EXPTIME-complete unlike the polynomial complexity for the model checking problem. The reason for this complexity may be intuitively seen from the above equation since the proof has to be carried out for all possible environments. This is unlike model checking, where the system is closed and hence the reachability proof proceeds without considering any nondeterminism in the environment.

While this sounds like bad news, a practical implementation of module checking is possible by the following observation.

Proceeding further, from Equation 1 we state that

\[ M \not|_o \varphi \iff \exists E' \cdot (M \times E' \not|_o \varphi \iff M \times E' \models \neg \varphi) \]  

In the above, the negation on \( \varphi \) can be pushed inside the formula such that all temporal and boolean operators are free from negation. Furthermore as the existence of \( E' \) ensures that \( M \not|_o \varphi \), \( E' \) acts as a witness to the violation. Such a witness can be used to provide useful insights to the reason for violation of a desired property over an open systems. The main beauty of this procedure is that we are looking for only one environment using which we can prove that the formula is not satisfied. We will illustrate later that we can employ a set of tableau rules to perform the computation of \( E' \) locally. This local computation is similar to on-the-fly model checking [1] where the state space of the system under verification is constructed on a need-driven basis. Even though the worst case complexity of local module checking is still bounded by the results of [10], we demonstrate that the proposed approach to local module checking yields much better practical results. The main contributions of this paper are:

(i) We propose a set of sound and complete tableau rules for local module checking. The proposed approach determines a single witness (environment) so that under the witness the negation of the given CTL property is satisfied. The proof proceeds only along a local set of states that are needed for the generation of a witness.

(ii) We have developed a local module checker by extending NuSMV [2]. Local module checking has been compared with the generation of the maximal environment under which the given CTL property is satisfied to demonstrate the performance gain of the proposed approach. The benchmarks are examples from NuSMV with both environment and system states.

The rest of this paper is organized as follows. Section 2 presents fundamental theory. Section 3 formulates CTL local module checking and presents the tableau rules for both system and environment states. Section 4 describes the implementation of a local module checker and section 5 contains the results obtained from it. Concluding remarks follow in section 6.
2 Preliminaries

Kripke Structure.
System behavior is described using Kripke structure \( M = (S, s^0, \rightarrow, P, L) \), where \( S \) is the set of states, \( s^0 \in S \) is the start state, \( \rightarrow \subseteq S \times S \) is the set of transition relations, \( P \) is the set of propositions relevant to \( M \) and finally, \( L : S \rightarrow 2^P \) is the labeling function mapping each state to set of propositions.

Temporal logic: CTL.
Properties of the system are defined using branching time temporal logic CTL. A CTL formula is defined over a set of propositions using temporal and boolean operators as follows:

\[
\begin{align*}
\phi \rightarrow P & \mid \neg P \mid tt \mid ff \mid \phi \land \phi \mid \phi \lor \phi \mid \text{AX}\phi \mid \text{EX}\phi \mid \text{A}(\phi\cup\psi) \mid \text{E}(\phi\cup\psi) \mid \text{AG}\phi \mid \text{EG}\phi
\end{align*}
\]

Note that CTL in general allows negations on temporal and boolean operators. However, we restrict ourselves to verifying CTL formulas where negations can only be applied to propositions.

Semantics of CTL formula, \( \varphi \) denoted by \( \langle\varphi\rangle_M \) is given in terms of set of states in Kripke structure, \( M \), which satisfies the formula. See Fig. 2.

\[
\begin{align*}
[p]_M &= \{s \mid p \in L(s)\} \\
[\neg p]_M &= \{s \mid p \not\in L(s)\} \\
[tt]_M &= S \\
[ff]_M &= \emptyset \\
[\varphi \land \psi]_M &= [\varphi]_M \cap [\psi]_M \\
[\varphi \lor \psi]_M &= [\varphi]_M \cup [\psi]_M \\
[\text{AX}\varphi]_M &= \{s \mid \forall s \rightarrow s' \land s' \in [\varphi]_M\} \\
[\text{EX}\varphi]_M &= \{s \mid \exists s \rightarrow s' \land s' \in [\varphi]_M\} \\
[\text{A}(\varphi\cup\psi)]_M &= \{s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \land \exists i, j. s_j \models \psi \land \forall i < j. s_i \models \varphi\} \\
[\text{E}(\varphi\cup\psi)]_M &= \{s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \land \exists i, j. s_j \models \psi \land \forall i < j. s_i \models \varphi\} \\
[\text{AG}\varphi]_M &= \{s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \land \forall i. s_i \models \varphi\} \\
[\text{EG}\varphi]_M &= \{s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \land \forall i. s_i \models \varphi\}
\end{align*}
\]

Fig. 2. Semantics of CTL

A state \( s \in S \) is said to satisfy a CTL formula \( \varphi \), denoted by \( M, s \models \varphi \), if \( s \in [\varphi]_M \). We will omit \( M \) from \( \models \) relation and \( [\varphi] \) if the model is evident in the context. We will also say that \( M \models \varphi \) iff \( M, s^0 \models \varphi \). The complexity for
model checking $M$ against a CTL formula $\varphi$ is $O(|M| \times |\varphi|)$ where $|M|$ and $|\varphi|$ are size of the model and the formula respectively.

**Module Checking.** [8]

In contrast to model checking where all transitions in every state of the model are always enabled, module checking is specifically directed for verification of models of open systems with states where the environment decides which transitions are enabled. Typically, in models of open systems, modules in short, the states are partitioned into two sets $S_s$ and $S_e$ where $S_s$ consists of system states with all outgoing transitions enabled while $S_e$ is the set of environment-controlled states where some (at least one) transitions are enabled. Note that, the environment can enable one or more transitions but cannot disable all transitions.

[10] presents the complexities for module checking in the setting of different temporal logic. While LTL module checking problem has the same complexity as LTL model checking, module checking branching-time temporal logics (CTL, CTL*) is harder compared to corresponding model checking. It has been proved that the problem of module checking is EXPTIME-Complete for CTL and 2EXPTIME-complete for CTL* specifications.

### 3 Tableau-based CTL Module Checking

In this paper, we present a technique for module checking CTL specifications such that the state-space of the module is explored locally and on-the-fly, i.e. our technique only explores the states needed to (dis)satisfy a given CTL formula.

We consider the behavior of a module $M$ in the context of an environment $E$ such that at each system state of $M$, the environment is incapable of altering the behavior of the module while at each environment state, the environment can decide which transitions to enable. To address such restrictions on the interaction, the behavioral patterns of $M$ and $E$ are described using labeled Kripke structure defined as follows:

**Definition 3.1** [Labeled Kripke Structure] A labeled Kripke structure $LKS = (S, s^0, \rightarrow, P, L, K)$ where $S, s^0, P, L$ are defined as before, $K$ is the maximum out-going branching factor of states in $S$ and $\rightarrow \subseteq S \times \{1, 2, \ldots, n\} \times S$ with $n \leq K$.

The state set $S$ of a module may be partitioned into two subsets, $S^e$, the set of all environment states and $S^s$, the set of all system states. In the above, each transition is annotated by a branching identifier whose domain is equal to the maximum branching factor. We will write $s \xrightarrow{i} s'$ to denote the $i$-th out-going transition from $s$ if $(s, i, s') \in \rightarrow$.
Definition 3.2 [Parallel Composition] Given a module \( \mathcal{M} = (S_M, s^0_M, \rightarrow_M, P_M, L_M, K) \), its environment \( \mathcal{E} = (S_E, s^0_E, \rightarrow_E, P_E, L_E, K) \), their parallel composition resulting in \( \mathcal{M} \times \mathcal{E} \equiv \mathcal{P} = (S_P, s^0_P, \rightarrow_P, P_P, L_P, K) \) are defined as follows:

(i) \( S_P \subseteq S_M \times S_E \)
(ii) \( s^0_P = (s^0_M, s^0_E) \)
(iii) \( P_P = P_M \cup P_E \)
(iv) \( L_P(s_1, s_2) = L_M(s_1) \cup L_E(s_2) \) where \( s_1 \in S_M \) and \( s_2 \in S_E \)
(v) \( (s_1, s_2) \stackrel{i}{\rightarrow} P \) \((s'_1, s'_2)\) if \( s_1 \stackrel{i}{\rightarrow} M s'_1 \) and \( s_2 \stackrel{i}{\rightarrow} E s'_2 \) where \( s_1, s'_1 \in S_M \) and \( s_2, s'_2 \in S_E \)

Furthermore, following constraints are imposed to restrict \( \mathcal{E} \)

(a) System-state conformity. If \( s_1 \) is a system state then \( \forall s_1 \stackrel{i}{\rightarrow} M s'_1 \Rightarrow \exists s_2 \stackrel{i}{\rightarrow} E s'_2 \) and vice versa.

(b) Environment-controllability. If \( s_1 \) is an environment-controlled state then \( \exists s_2 \stackrel{i}{\rightarrow} E s'_2 \land (\forall s_2 \stackrel{i}{\rightarrow} E s''_2 \Rightarrow \exists s_1 \stackrel{i}{\rightarrow} M s''_1) \)

Given a CTL formula \( \varphi \), we say that \( \mathcal{M} \times \mathcal{E} \equiv \mathcal{P} \models \varphi \iff s^0_P \models \varphi \).

Module checking, denoted by \( \mathcal{M} \models_o \varphi \), therefore, amounts to deciding whether \( \forall \mathcal{E}. \mathcal{M} \times \mathcal{E} \models \varphi \); i.e.

\[ \mathcal{M} \models_o \varphi \iff \forall \mathcal{E}. \mathcal{M} \times \mathcal{E} \models \varphi \quad (3) \]

Proceeding further, from Equation 3 we state that

\[ \mathcal{M} \not\models_o \varphi \iff \exists \mathcal{E}' (\mathcal{M} \times \mathcal{E}' \not\models \varphi \iff \mathcal{M} \times \mathcal{E}' \models \neg \varphi) \quad (4) \]

It is important to note however that \( \mathcal{M} \not\models_o \varphi \not\iff \mathcal{M} \models_o \neg \varphi \). It can be shown that a module does not satisfy both a formula and its negation\(^4\). A module might satisfy a formula and its negation under different environments.

For example: given \( \mathcal{M} = (\{s^0, s_1, s_2\}, s^0, \{s^0 \rightarrow s_1, s^0 \rightarrow s_2, s_1 \rightarrow s_1, s_2 \rightarrow s_2\}, \{p\}, L, 2) \) where \( L(s_1) = \{p\} \) and CTL formula \( AXp \), it is easy to see that \( \mathcal{M} \not\models_o AXp \) and \( \mathcal{M} \not\models_o EX\neg p \).

3.1 Local Module Checking and Generation of Witness

We present here a tableau-based technique similar to [4] for constructing the witness environment \( \mathcal{E} \), existence of which ensures that the module does not satisfy original formula. Tableau rules are defined as

\[
\begin{array}{c|c}
\text{Antecedent} & \text{Consequent} \\
\hline
\end{array}
\]

where the Antecedent represents the current obligation for module checking and Consequent denotes the next obligation. A successful tableau (see below)

\(^4\) Note that the same is not true for model checking problem: \( M \not\models \varphi \not\iff M \models \neg \varphi \)
will result in automatic generation of the environment $E$. Figs. 3 and 4 present the complete tableau where the former corresponds to the rules for system
basu, roop and sinha

states and the latter for the environment-controlled states.

**Tableau for System States.** Consider first the Fig. 3 (without the reorg rule).

The rules for prop, ∧, ∨ are simple and intuitive. The prop rule leads to a successful tableau leaf, the ∧ rule is successful if both its consequents are successful and finally, the success of ∨-rule depends on the success of any of its consequents. The rule unr,eu corresponds to unrolling of the EU formula expression. A state satisfies $E(\varphi U \psi)$ iff (a) $\psi$ is satisfied in the current state or (b) $\varphi$ is true in the current state and in one of its next states $E(\varphi U \psi)$ is satisfied. The rule for unr,au is similar to unr,eu with exception of the presence of universal quantification on the next states (AX). The rule unr,eg (unr,ag) states that the current state satisfies $\varphi$ and in some (all) next state EG$\varphi$ (AG$\varphi$) holds true.

Finally, the rules for unr,s,ex and unr,s,ax correspond to the unfolding of the state and the formula expression simultaneously. Note that in the former, we are searching for at least one next state, while in the latter all next states should satisfy $\varphi$. As such for the EX-formula expression, the tableau selects any one of the next states $s_j|e_i$ and if the selected state satisfies $\varphi$, there is no obligation left for the rest of the next states; the obligations on the remaining next states $s_j,...,k$ in the context of the environment is to satisfy $tt$ (any state satisfies the propositional constant $tt$). Note that, the tableau can potentially have $k$ sub-tableaus for unr,s,ex each of which will correspond to selection of one next state $s_i$ from the set of $k$ next states of $s_s$.

Observe that, the rules unr,s,ex, unr,s,ax lead to one-step construction of the environment. Conforming to the constraint that the environment at the system state cannot control the enabled transition, the environment must have exactly the same number of transitions as the system state.

**Tableau for Environment-Controlled States.** The tableau rules (Fig. 4) for environment controlled states are slightly different from the one described above. Instead of asking whether a state satisfies a formula expression (see Fig. 3), the question asked is whether a state satisfies a set of formula expressions. In fact the set represents a formula expression equivalent to conjunction over its elements. The reason for altering the tableau rule structure stems from the fact the environment plays an active role in deciding the enabled transitions. For example: in order to construct an environment state $e$ such that $s_e|e| = \varphi \land \psi$, we need to construct $e_1$ and $e_2$ such that $s_e|e_1| = \varphi$ and $s_e|e_2| = \psi$ with the constraint that $e_1 = e_2$, i.e. exactly the same set of transitions is enabled to module check $\varphi$ and $\psi$ at the state $s_e$. To address to this state of affairs, the tableau rules for environment-controlled state maintains a global view of all the formula to be satisfied and ensures consistent enabling/disabling of transitions by the environment (to be constructed).

---

5 For the purpose of constructing the environment, in Rule unr,s,ex, we can safely assume that all the environment states $e_j,...,k$ replicates the behavioral patterns of module states $s_j,...,k$. 

131
The rules for \( \text{prop}, \lor, \text{unr}_{eu}, \text{unr}_{au}, \text{unr}_{eg}, \text{unr}_{ag} \) in Fig. 4 are similar to that in Fig. 3. The rule for \( \land \) aggregates all the conjuncts in the set. The \text{emp}-rule represents the case the state does not have obligation to satisfy any formula and a (successful) tableau leaf is reached. The rule \text{unr}_{ex} is applied only when no other rules are applicable. In other words, the set \( C \) only contains \( EX \) and/or \( AX \) formula expressions. \( C_{ax} \) is the set of all formula expressions that must be satisfied in all next states while \( C_{ex} \) is the set of the formula expressions each of which must be satisfied in at least one of the next states.

\( \Pi \) records all the indices of the outgoing transitions from \( s_e \), while \( \Pi_{C_{ax}}(\pi) \) is a subset of \( C_{ex} \) such that there is at least one subset for each \( i \) present in a subset \( \pi \in \Pi \). For example, if \( \pi \) is a singleton set, then \( \Pi_{C_{ex}}(\pi) \) is also a singleton set containing \( C_{ex} \). In short, \( \Pi_{C_{ex}}(\pi) \) is used to associate with \( i \)-th selection of next state-environment pair a set of elements \( C_i \subseteq C_{ex} \). The consequent of the rule, therefore, fires the obligation that all states identified by the indices in \( \pi \) must satisfy \( C_{ax} \) and the corresponding subset of \( C_{ex} \) as identified by \( \Pi_{C_{ex}}(\pi) \).

This rule is illustrated by Fig. 5. All states in the Fig. 5(a) are environment states and the proposition \( p \) is true at states \( s_1, s_4, \) and \( s_6 \). The obligation at \( s_0|e_0 \) is to satisfy \( C = \{ AXEXp, EXEX\neg p, EXp \} \). As there are 3 transitions from \( s_0 \) there are \( 2^3 - 1 = 7 \) different choices for \( \pi \). Fig. 5(b) shows subsets consisting of only 1 and 2. \( \pi \) represents the indices of enabled transitions. These transitions lead to states which must satisfy all the elements of \( C_{ax} = \{ EXp \} \). Corresponding to each \( \pi \), there exists at least one choice for \( \Pi_{C_{ex}}(\pi) \) which subsets \( C_{ex} = \{ p, EX\neg p \} \) in \( |\pi| \) subsets where \( |\pi| \) is the size of \( \pi \). It also assigns each subset to different subset of next states where elements of the assigned subset must be satisfied. For example for \( \pi = \{ 1, 2 \} \), there are two possible ways of assigning subsets of \( C_{ex} \) to \( s_1|e_1 \) and \( s_2|e_2 \) (see Fig. 5(b)). In this example, we obtain an environment \( e_0 \) for \( \pi = \{ 1, 2 \} \) (i.e. the transition labeled 3 from \( s_0 \) is disabled), and \( \Pi_{C_{ex}}(\pi) = \{ \{ p \}, \{ EX\neg p \} \} \). Note that, our local approach does not necessarily examine all possible choices for \( \pi \) and the corresponding subsets \( \Pi_{C_{ex}}(\pi) \); instead it terminates as soon as an environment that leads to satisfiability of given formula is obtained.

Finally, consider the \text{reorg} (re-organize) rules in Figures 3 and 4. These rules rearrang the formula obligations from set-based to expression-based or vice versa depending on the type of the model state being considered.

\textit{Finitizing the Tableau.} The given tableau rules can be of infinite depth as each recursive formula expressions \( AU, EU, AG, EG \) are unfolded infinitely many times. However, the total number of states in the Kripke structure for the module is \( N = |S| \), and this finitizes the tableau depth. In Fig. 3, if the pair \(( s_s, \varphi ) \) in \( s_s|e = \varphi \), where \( \varphi \) is either \( EG \) or \( AG \) formula expression, appears twice in a tableau path, we fold back the tableau by pointing the second occurrence to the first and stating a successful tableau loop is obtained. Note that this also leads to generation of a loop in the constructed environment. On the other hand, if the pair \(( s_s, \varphi ) \), where \( \varphi \) is of the form \( EU \) or \( AU \),
The idea of folding back or replacing using false relies on fixed point semantics of CTL formulas. The CTL formulas $\text{EG}$ and $\text{AG}$ can be represented by greatest fixed point recursive equations:

$$\text{EG} \varphi \equiv Z = \nu (\varphi \land EXZ)$$
$$\text{AG} \varphi \equiv Z = \nu (\varphi \land AXZ)$$

In the above $\nu$ represents the sign of the equation and is used to denote greatest fixed point and $Z$ is recursive variable whose valuation/semantics (set of model states) is the greatest fixed point computation of its definition. Similarly the fixed point representation of CTL formulas $\text{AU}$ and $\text{EU}$ are

$$E(\varphi U \psi) \equiv Z = \mu \psi \lor (\varphi \land EXZ)$$
$$A(\varphi U \psi) \equiv Z = \mu \psi \lor (\varphi \land AXZ)$$

The fixed point computation proceeds by iteratively computing the approximations of $Z$ over the lattice of set of states in the model. A solution is reached only when two consecutive approximations are identical. For greatest fixed point computation, the first approximation is the set of all states (top of the lattice) and as such a system can satisfy a greatest fixed point formula along an infinite path (using loops). On the other hand, the first approximation of the least fixed point variable is empty set (bottom of the lattice) and therefore satisfiable paths for least fixed point formula are always of finite length.

For the tableau in Fig. 4, the finitization condition is similar. If the pair $(s_e, C)$ in $s_e \models C$ appears twice in a tableau path and if $C$ contains any least fixed point CTL formula expression, then the second occurrence is replaced by $ff$; otherwise the second occurrence is made to point to the first and a successful tableau is obtained.

Theorem 3.3 (Sound and Complete) Given a module $\mathcal{M}$ and CTL formula $\varphi$, $\mathcal{M} \not\models_o \varphi$, iff the tableau in Figures 3 and 4 generates an environment $\mathcal{E}$ where $\mathcal{M} \times \mathcal{E} \models \neg \varphi$. 

133
Proof.

The proof proceeds by realizing the soundness and completeness of each of the tableau rules. For brevity, we present here the proof-sketch for \textbf{unr}_{se}, proofs for the other rules are straightforward.

Recall that, $s_e \models e \models C$, where $C$ is the set of formula expressions with temporal operators $AX$ and $EX$, is satisfiable if the next states proof obligations are satisfied by destination states reachable via transitions enabled by the environment $e$. The environment can enable any subset (barring $\emptyset$) of transitions. The tableau rule, therefore, considers all possible subset of destination states of enabled transitions.

As each transition is annotated by an index (whose domain is over the outgoing branching factor of $s_e$), we construct $\Pi$, the set of indices of outgoing transitions. In other words, $i \in \Pi \Rightarrow s_i$ is reachable via the transition with index $i$. We are required to identify one possible subset of $\Pi$ which represents the enabled transitions whose destinations conform to the satisfiability obligations in the consequent (see $\exists \pi \subseteq \Pi$ in the consequent). Let $\pi = \{s_i \mid i \in \pi\}$ be the next states reachable via (selected) enabled transitions.

The consequent of the tableau rule has the following obligations. All elements of $\pi_s$ in parallel composition with the environment must satisfy the expressions in $C_{ax}$ and for each formula expression $\varphi$ in $C_{ex}$, there must be at least one state which in conjunction with the corresponding environment satisfies $\varphi$. Observe that, there is requirement for an existence of a subset, $\Pi_{C_{ex}}(\pi)$, of $C_{ex}$ corresponding to a subset $\pi$ such that next state-environment pairs satisfy the corresponding obligations. This ensures that the environment constructed is consistent, i.e., $e_i$ is constructed such that $s_i | e_i$ satisfies both the for all obligations ($C_{ax}$) and its share of existential obligations ($C_i$). Therefore, if we can generate an environment for $s_e$ corresponding to rule \textbf{unr}_{se}, then $s_e \not\models_o \psi$ where $\psi$ is the disjunction of the elements of the set $\{AX\neg \varphi \mid \varphi \in C_{ex}\} \cup \{EX\neg \psi \mid \psi \in C_{ax}\}$. The other direction can be proved likewise. \hfill $\square$

Complexity.

The main factor in complexity is attributed to the handling of universal and existential formulas in \textbf{unr}_{se} rule in Fig. 4. The number of different obligations that can be fired on the basis of each selection of $\pi \subseteq \Pi$, is $O(2^{\lvert C_{ex}\rvert \times \lvert \pi \rvert})$ where $\lvert C_{ex}\rvert$ and $\lvert \pi \rvert$ are respectively the size of the respective sets. This is because we need to consider all possible permutations of elements of $C_{ex}$ and match the permutations with all possible permutations for selecting element from $\pi$. Overall complexity is, therefore, exponential to the maximum branching factor (maximum size of $\pi$) of the module times the size of the formula to be satisfied. It is worth noting here that if the given formula is free from $EX$ and $EU$, the complexity of tableau-based approach will be polynomial to the sizes of the formula and module (same as model checking). This is due to the fact that in the presence of only $AX$-formulas in rule \textbf{unr}_{se}, we are only required
to find one element from $\Pi$; the state corresponding to which satisfies $C_{ax}$.

Figure 6 shows the witness environment generated for the coffee brewer in Figure 1. The witness environment disables three transitions of state $s_1$ such that there is no path where ten cups can be selected. Under this witness, it can be shown that the model does not satisfy the original property $AG (tt U TEN \land AF (SERVE \lor ERROR))$.

4 Implementation

A local module checking tool has been implemented using C/C++ and many SMV benchmarks from the NuSMV package have been tested. The tool proceeds as follows:

(i) An SMV model is converted into an explicit state FSM using NuSMV. This is achieved by traversing the model’s state space in NuSMV and writing all reachability information (states, transitions and labels) to a file. As there is no explicit notion of system/environment states in NuSMV, the state space is divided randomly into two sets of equal size representing system and environment states respectively.

(ii) The file containing the explicit state FSM is read by the tool along with the CTL property to be used for verification.

(iii) The CTL property is negated and the negation is carried inwards. CTL properties can not have negations applied to formulas other than propositions.

(iv) A search for a witness is carried out, starting with the initial state, and the tool attempts to generate an environment under which the module satisfies the negated CTL property. If a witness is present, then the module does not satisfy the original property.
The algorithm applies the appropriate system or environment state tableau rules on the current state. Once all present (current-state) commitments are met, any future commitments are passed to its successors. It uses a heuristic to compute a small set of successors of the current state which satisfy all its future commitments. This is used to ensure that the generated witness is small. If no witness can be computed, it can be concluded that the module satisfies the CTL property. Note that the algorithm attempts to generate a small witness and not the minimal witness for a given property and module. In order to compare the obtained results, the tool was extended to find an environment under which the original CTL property is satisfied. This is achieved as follows:

(i) The CTL property (non-negated) is read along with the module FSM (as above).

(ii) The algorithm attempts to find an environment under which the given CTL property is satisfied by traversing all of the reachable state space.

It is important to note here that the above is not an implementation of global module checking. Global module checking advocates the need to check if the given property is satisfied by the module under all environments. However, the above approach constructs only a single environment under which the given CTL property is satisfied by the module. However, unlike the local module checker which attempts to generate a small witness, the algorithm for global module checking constructs the biggest environment under which the module satisfies the given property. This is done by enabling all but those transitions in reachable environment states which may lead to the dissatisfaction of the given property. It was observed that on the average, computing the maximal environment under which the original CTL property is satisfied consumes more time than computing a small witness under which it fails.

The problem of computing all possible environments (global module checking) is even more harder and time consuming and the benefits of local module checking will be even more apparent in such a case.

5 Experimental Results

The results are given in table 1. The first column contains the name and size (in number of states) of the verified module and the CTL property used is given in the second column. The results obtained from local module checking are presented in the third column. The fourth column contains the results of generating the maximal environment under which the given CTL property is satisfied.\(^6\)

Note that for many modules, the original property and its negation were both satisfied under different environments. A majority of models had multiple start states. In these cases, the local module checker (and the maximal witness

---

\(^6\) The results are in format TimeTaken(seconds)/Result(SUCCESS or FAILURE)/Number of disablings (Number of states traversed locally during local module checking).
### Table 1
**Implementation Results**

| System($|S|$) | CTL Property | Local module checking | Generation of maximal witness |
|-----------------|-----------------|------------------------|-------------------------------|
| short(4)        | $AG(\text{request} \rightarrow AF \text{state} = \text{busy})$ | 0.02/F/0 (2) | 0.05/S/0 |
| ring(7)         | $(AG AF \text{gate1.output}) \land (AG AF \text{gate1.output})$ | 0.01/F/0 (2) | 0.02/S/0 |
| Counter(8)      | $AG(\text{bit2.value} = 0)$ | 0.00/S/0 (5) | 0/F/0 |
| coffee(10)      | $AG AF (\text{ten} \land EF \text{Serve})$ | 0.008/S/3 (7) | 0.001/S/0 |
| MCP(30)         | $AGEF((\text{MCP.missionaries} = 0) \land (\text{MCP.cannibals} = 0))$ | 0.001/S/5 (2) | 0.05/S/0 |
| base(1758)      | true $AU(\text{step} = 0)$ | 1.250/F/0 (21) | 1.290/S/0 |
| periodic(3952)  | $AG(\text{aux} \neq p11)$ | 9.580/S/0 (701) | 51.270/F/0 |
| syncarb5(5120)  | $AGEF_{\text{e5.\textbf{Persistent}}}$ | 7.73/S/0 (223) | 3704/S/960 |
| dme1(6654)      | $AG((e - 1.r.out = 1) \land (e - 2.r.out = 1) \land (e - 1.r.out = 1) \land (e - 3.r.out = 1) \land (e - 2.r.out = 1) \land (e - 3.r.out = 1))$ | 5.490/F/0 (141) | 41.17/S/790 |
| pqueue(10000)   | $E G(\text{out}[1] = 0)$ | 34.20/F/0 (1904) | 35.130/S/0 |
| pqueue(10000)   | $A F(\text{out}[1] = 0)$ | 34.930/F/0 (2101) | 34.960/S/0 |
| barrel(45025)   | $AG true AU (b0 = 0)$ | 12.720/S/0 (234) | 34.190/S/1 |
| idle(87415)     | true $AU(\text{step} = 0)$ | 77.190/F/0 (38088) | 79.920/S/0 |
| abp4(275753)    | $EF(\text{sender.state} = \text{get})$ | 130.77/F/0 (59808) | 133.880/S/0 |

**6 Conclusions**

Module checking extends model checking for open systems. It has been shown in [10] that the complexity of module checking for branching time logic CTL is EXPTIME complete. The above approach to module checking generates all possible environments so that the composition of the model and the environment satisfies the CTL property. In this paper, we propose a local approach to CTL module checking. The proposed approach tries to determine a witness environment so that the negation of the property is satisfied by the composition of the witness and the model. When this is possible, the original property is not satisfied over the module. We have developed a set of sound and complete tableau rules for local module checking of CTL. The efficiency of the proposed approach is demonstrated by comparing the performance of local and traditional global module checking using benchmarks from NuSMV. The results presented compare the generation of one environment in both cases.
Answering the module checking question using a global strategy requires the generation of all environments, which will be computationally much more expensive than the local approach.

References


Appendix

Example.

Table 2 describes the steps involved in generating a witness environment for the coffee brewer example in Figure 1. First, the original CTL property $\text{AG} (tt \text{UTEN} \land \text{AF} (\text{SERVE} \lor \text{ERROR}))$ is negated (and the negation carried inwards) to $\text{E} (tt U \text{AG} \neg \text{TEN} \lor \text{AG} (\neg \text{SERVE} \land \neg \text{ERROR}))$. The local module checker is then called on the initial state ($s_0$) of the model along with the negated formula. The algorithm applies the appropriate tableau rules described earlier and first attempts to check if the current state satisfies all current-state commitments. Then the next-state commitments are passed on to the successors.

For example, initially the negated property $\text{E} (tt U \text{AG} \neg \text{TEN} \lor \text{AG} (\neg \text{SERVE} \land \neg \text{ERROR}))$ is passed to the initial state $s^0$ of the model. The negated property is then broken down (using the environment state tableau rule unr$_{eu}$) to $\{\text{AG} (\neg \text{TEN} \lor \text{AG} (\neg \text{SERVE} \land \neg \text{ERROR})) \lor (tt \land \text{EX} tt U \text{AG} (\neg \text{TEN} \lor \text{AG} (\neg \text{SERVE} \land \neg \text{ERROR})))\}$. The resulting disjunction is then broken further (using the tableau rule $\land$). Once the current-state commitments are met, all next state commitments of $s^0$ are passed to its successor $s_1$. This is shown using $\lor$ in Table 2. Note that for environment states, formulas are organized into set notation whereas for system states, they are applied in the order they arrive. The algorithm terminates when a strongly connected component which satisfies the negated property is found.
\[ s^0 | e_0 | = \{ E(tt U AG \neg TEN \lor AG(\neg SERVE \land \neg ERROR)) \} \]
\[ s^0 | e_0 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \lor (tt \land EXE(tt U AG \neg TEN \lor AG(\neg SERVE \land \neg ERROR))) \} \]
\[ s^0 | e_0 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s^0 | e_0 | = \{ \neg TEN, AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_1 | e_1 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_1 | e_1 | = \{ \neg TEN, AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_2 | e_2 OR s_3 | e_3 OR s_4 | e_4 OR s_5 | e_5 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_2 | e_2 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \} \ (s_3, s_4, s_5 \text{ not explored}) \]
\[ s_2 | e_2 | = \{ \neg TEN, AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_6 | e_6 | = AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \]
\[ s_6 | e_6 | = \neg TEN \land AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \]
\[ s_7 | e_7 | = AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \]
\[ s_7 | e_7 | = \neg TEN \land AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \]
\[ s_8 | e_8 AND s_9 | e_9 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s_8 | e_9 AND s_9 | e_9 | = \{ \neg TEN, AXAG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]
\[ s^0 | e_0 | = \{ AG(\neg TEN \lor AG(\neg SERVE \land \neg ERROR) \} \]

Table 2
Tableau for constructing witness environment for the coffee brewer example in Figure 1

140
Specification and Generation of Environment for Model Checking of Software Components

Pavel Parizek\textsuperscript{a,1}, Frantisek Plasil\textsuperscript{a,b,1}

\textsuperscript{a} Department of Software Engineering
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
\{parizek, plasil\} \@ nenya.ms.mff.cuni.cz

\textsuperscript{b} Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic
plasil \@ cs.cas.cz

Abstract
Model checking of isolated software components is inherently not possible because a component does not form a complete program with an explicit starting point. To overcome this obstacle, it is typically necessary to create an environment of the component which is the intended subject to model checking. We present our approach to automated environment generation that is based on behavior protocols [9]; to our knowledge, this is the only environment generator designed for model checking of software components. We compare it with the approach taken in the Bandera Environment Generator tool [12], designed for model checking of sets of Java classes.

Key words: Software components, behavior protocols, model checking, automated generation of environment

1 Introduction
Model checking is one of the approaches to formal verification of software systems that gets a lot of attention at present. Still, there are some obstacles that have to be addressed, at least partially, before model checking of software can be widely used in practice. Probably the biggest problem is the size of state space typical for software systems. One solution to this problem (state explosion) is the decomposi-tion of a software system into small and well-defined units, components.

\textsuperscript{1} This work was partially supported by the Czech Academy of Sciences (project 1ET40030504) and France Telecom under the external research contract number 46127110.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Nevertheless, a component usually cannot be checked in isolation, because it does not form a complete program inherently needed to apply model checking. It is, therefore, necessary to create a model of the environment of the component subject to model checking, and then check the whole program, composed of the environment and component. The environment should be created in a way that minimizes the increase of the state space size caused by the composition.

1.1 Goals and Structure of the Paper

The paper aims at addressing the problem of automated generation of environment for model checking of software components implemented in the Java language. The main goal is to present our approach that is based on behavior protocols [9] and to compare it with the approach taken in the Bandera Environment Generator tool [12], which is the only other Java focused approach we are aware of.

The remainder of the paper is organized as follows. Sect. 2 provides an example to illustrate the problem of environment generation and Sect. 3 introduces the Bandera Environment Generator (BEG) [12]. Sect. 4 starts with an overview of behavior protocols [9] and then presents the key contribution - the description of our approach to specification and generation of environment based on behavior protocols. Sect. 5 provides comparison of the two approaches and briefly mentions our proof of concept implementation. The rest of the paper contains related work and a conclusion.

2 Motivation

In order to illustrate how an environment can be created, we present a simple example - a Java class DatabaseImpl and a handwritten environment for this class, assuming DatabaseImpl is the intended subject to model checking. The class implements one interface and requires one internal reference of an interface type to be set. Therefore, it can be also looked upon as a Database component with one provided and one required interface.

Key fragments of source code of the DatabaseImpl class look as follows:

```java
public interface IDatabase {
    public void start();
    public void stop();
    public void insert(int key, String value);
    public String get(int key);
}

public class DatabaseImpl implements IDatabase {
    private ILogger log;
```
public class EnvThread extends Thread {
    IDatabase db;

    public void run() {
        db.insert(getRandomInt(), getRandomString());
        String val = db.get(getRandomInt());
        ...
    }
}

public static void main(String[] args) {
    IDatabase db = new DatabaseImpl();
    db.setLogger(new LoggerImpl());

    db.start();

    new EnvThread(db).start();
    new EnvThread(db).start();
    ...

In the example, two threads of control, which enable the model checker to search for concurrency errors, are created. A random choice of parameter values for the purpose of checking all the control flow paths is employed as well (getRandom... calls).

Obviously, creating an environment by hand is hard and tedious work even in simple cases. A straightforward solution to this problem is to automatically generate the environment from a higher-level abstraction than the code provides. In Sect. 3 and 4, we present two solutions based on this idea.

3 Environment Generator in Bandera

3.1 Bandera

Bandera [6] is a tool set designed for model checking of complete Java programs, i.e. those featuring a main method. It is composed of several modules - model extractor, model translator, environment generator, and model checker, to name the key of them. The model extractor extracts a (finite) internal model from Java source code and the model translator translates the internal model into the input language of a target model checker. Here, the Bandera tool set supported the Spin and Java PathFinder model checkers originally, but currently it is intended mainly for a Bandera specific model checker (Bogor [11]).

3.2 Bandera Environment Generator

The Bandera Environment Generator (BEG) [12] is a tool for automated generation of environment for Java classes. Given a complete Java program, the user of the BEG tool has to decompose the program into two parts - the tested unit, i.e. the classes to be tested, and its environment. Since the environment part is usually too complex for the purpose of model checking, it is necessary to create an abstract environment. This abstract environment can be generated from a model created

- either from assumptions the user provided, or
- from a result of code analysis of environment classes (if available).

The model can specify, for example, that a certain method should be called five times in a row, or that it should be executed in parallel with another specific method.

Since, usually, there exist no environment classes in case of software components, we will further consider only the first option - i.e. that the abstract environment is generated from user-specified assumptions. For this purpose, the BEG tool provides two formal notations - LTL and regular expressions.
The actual specification ("environment specification" in the rest of this section) takes the form of program action patterns (method calls, assignments, etc), illustrated below.

An environment specification for the DatabaseImpl class presented in Sect. 2, written in the input language of the BEG tool, could be as follows:

```plaintext
environment
{
  instantiations
  {
    1 LoggerImpl log;
    IDatabase db = new DatabaseImpl();
    db.setLogger(log);
    int x = 5;
  }

  -- high level specification of the environment behavior
  regular assumptions
  {
    T0: (db.get() | db.insert())*
    T1: (db.get(x) | db.insert(5, "abcd"))*
  }
}
```

The instantiations section allows the user to specify how many instances of a certain type should be created and under which names they can be referenced. In this example, two objects are instantiated - the log instance of the LoggerImpl class and the db instance of the DatabaseImpl class.

The regular assumptions section contains regular expressions describing the behavior of the environment with respect to the tested classes. Each regular expression defines a sequence of actions that should be performed by a single thread of control. In our example, two threads of control are defined, both modeling a sequence of calls to the insert and get methods on the IDatabase interface.

Notice that the whole execution is characterized by the specified threads (T0, T1) - there is no "main" thread. Consequently, calls to the start and stop methods on the IDatabase interface cannot be reasonably modeled in such an environment specification.

The BEG tool also allows to specify parameter values of method calls on the tested classes. If the value of a parameter is not specified, as in the thread T0 above, then it is non-deterministically selected from all the available values of a given type (e.g. from all allocated instances of a given class in the case of a reference type) during model checking. As a parameter to a method call, it is even possible to use a variable defined in the instantiations section (such as x above).
As the BEG tool is not intended specifically for software components, but rather for plain Java classes, it is necessary to manually specify the environment for the classes that implement a target component; an alternative would be to develop a tool for automatic translation of an ADL specification of the component’s architecture and behavior into the input language of the BEG tool.

However, since the most recent Bandera release is an alpha version only [6], not being fully stable yet, we have decided to use the Java PathFinder model checker (JPF) [13]. Consequently, we faced the problem to create an environment generator, since none was available (BEG is not intended for components and, moreover, the latest Bandera version does not allow to use the Java PathFinder as a target model checker any more).

4 Environment Generator for Java PathFinder

We have built our own environment generator for model checking of components implemented in the Java language. Our approach stems from the assumption that components are during design specified in an ADL (Architecture Description Language), which, in particular, includes specification of their provided and required interfaces and also specification of their behavior. The latter is done via behavior protocols [9]. In this section we show how this behavior specification can be advantageously employed for generating an environment necessary for component model checking.

4.1 Behavior protocols

A behavior protocol is an expression that describes the behavior of a software component in terms of atomic events on the provided and required interfaces of the component, i.e. in terms of accepted and emitted method call requests and responses on those interfaces.

![Diagram of DATABASE and LOGGER components](image)

Fig. 1. The DATABASE and LOGGER components, defined in Sect. 2

A protocol example for the Database component from Fig. 1 is below:

```plaintext
?db.start↑ ; !log.start ; !db.start↓ ; (?db.insert || ?db.get)* ; ?db.stop{!log.stop}
```
Since this protocol specifies the interplay on the external interfaces of Database, it is its frame protocol [9]. Informally speaking, it specifies the Database functionality that starts with accepting request for start call on db. As a reaction it calls start at log and issues response to the start call on db. This is followed by accepting insert on db in parallel with get on db finitely many times. At the end, it accepts a request for a stop call on db and, as a reaction, it calls stop at log and issues response to the stop call on db.

Each event has the following syntax: <prefix><interface>.<method> <suffix> (where the suffix is optional; the events having no suffix are syntactical shortcuts explained below). The prefix ? denotes an accept event and the prefix ! denotes an emit event. The suffix ↑ stands for a request (i.e. a method call) and the suffix ↓ stands for a response (i.e. return from a method). An expression of the form !i.m is a shortcut for !i.m↑;?i.m↓, an expression of the form ?i.m is a shortcut for ?i.m↑;!i.m↓ and an expression of the form ?i.m{prot} is a shortcut for ?i.m↑;prot;!i.m↓, where prot is a protocol. The NULL keyword denotes an empty protocol.

The example above presents also several operators. The ; character is the sequence operator, * is the repetition operator and || is the or-parallel operator. Behavior protocols support also an alternative operator + and an and-parallel operator |. In fact, the or-parallel operator is only a shortcut; e.g. a || b stands for a + b + (a | b). The | operator denotes all the possible interleavings of traces that correspond to its operands.

A behavior protocol defines a possibly infinite set of event traces, each of them being finite.

Each component has a frame protocol associated with it, and a composite component can have also an architecture protocol [9]. The frame protocol of a component describes its external behavior, what means that it can contain only the events on external interfaces of the component. On the other hand, the architecture protocol describes the behavior of a component in terms of composition of its subcomponents at the first level of nesting.

4.2 Cooperation of Java PathFinder with the Protocol Checker

When checking a component application specified via ADL with behavior protocols, it is necessary (i) for each composite component in the hierarchy to check compositional compliance of subcomponents at the first level of nesting and also compliance of a frame protocol with an architecture protocol (ii) and for each primitive component to verify that an implementation of the component obeys its frame protocol. For the purpose of checking compliance of protocols, we use the protocol checker [7] developed in our research group, and for checking that a primitive component obeys its frame protocol, we use a tool created via cooperation of JPF with our protocol checker [8]. The tool has to be applied to a program composed of a target component and its
Communication between JPF and the protocol checker during checking of the Database component is depicted on Fig. 2. The left part of the schema shows the JPF traversing the code (state space) of the component and the right part shows the state space of the protocol checker, which is determined by the frame protocol of the component. A plugin for JPF, which we have developed, traces execution of the invoke and return instructions that are related to methods of the provided and required interfaces of a target component, and notifies the protocol checker of those instructions in the form of atomic request and response events. The protocol checker verifies that the trace constructed from the received events is compliant with the frame protocol of the component. When the protocol checker encounters an unexpected event or a missing event, it tells JPF to stop the state space traversal and to report an error (counter example) to the user.

4.3 Modeling the Environment with Inverted Frame Protocol

The environment of a component can be advantageously modeled by its inverted frame protocol [1], constructed from the components frame protocol by replacing all the accept events with emit events and vice versa. The inverted frame protocol constructed this way forces the environment

- to call a certain method of a particular provided interface of the component at the moment the component expects it, and
- to accept a certain method call issued on a particular required interface of the component at the moment the component “wishes” to do so.

The inverted frame protocol of the Database component introduced above is:

\[
!\text{db.start}↑ ; ?\text{log.start} ; ?\text{db.start}↓ ; (!\text{db.insert} || !\text{db.get})^* ; !\text{db.stop}{?\text{log.stop}}
\]
Our environment generator accepts all syntactically valid frame protocols with the exception of protocols of the form \(?a + !b\) and \(!a* ; ?b\). The reason for not supporting frame protocols of the form \(?a + !b\) is that the environment driven by inversion of such a protocol cannot determine how long it should wait for the \(!b\) event to occur before it emits a call that corresponds to the \(?a\) event and therefore disables the other alternative (i.e. \(!b\)). Protocols of the form \(!a* ; ?b\) are not supported for a similar reason - the environment is not able to determine when the repetition \(!a*\) is going to finish. It is recommended to use protocols of the form \(!a* ; !b\) instead (wherever possible) because in such case the \(!b\) event tells the environment that the repetition has finished.

In order to minimize the size of the state space that JPF has to traverse, our environment generator performs several transformations of the frame protocol of the target component before creating the inverted frame protocol and generating the code of the environment. The key goal of the transformations is to

- get as many instances of the alternative operator \(+\) as possible at the outermost level of protocol nesting. The advantage of this approach is that all these alternatives can be checked in parallel by multiple instances of JPF, thus lowering the time requirements for model checking of the target component.
- reduce the number of repetitions, and also event interleavings caused by the \(|\) operator, even at the cost of accuracy.

For example, our generator transforms

- an iteration over some subprotocol to an alternative between an empty protocol and a sequence of two copies of the subprotocol (e.g. the protocol \(!a*\) is transformed to the protocol \(\text{NULL} + (!a ; !a)\)),
- a sequence that contains some alternatives to an alternative between all possible sequences (e.g. the protocol \(!a ; (!b1 + !b2)\) is transformed to the protocol \((!a ; !b1) + (!a ; !b2)\)),
- an and-parallel operator connecting two subprotocols, both of them being alternatives, to an alternative between selected pairs of subprotocols connected by the \(|\) operator - the pairs are selected in a way ensuring that each element of the two alternatives is present at least in one of the pairs (e.g. the protocol \((!a1 + !a2) | (!b1 + !b2)\) is transformed to the protocol \((!a1 | !b1) + (!a2 | !b2)\)), and
- an and-parallel operator with three or more subprotocols to an alternative between selected pairs of subprotocols, where each pair is connected by the \(|\) operator and followed by a sequence of subprotocols that do not belong into the selected pair; the pairs are selected in such a way that the first subprotocol is paired with the second, the second with the third, and so on (e.g. the protocol \(a | b | c | d\) is transformed to the protocol \(((a | b) ; c ; d) + ((b | c) ; a ; d) + ((c | d) ; a ; b))\)).
4.4 Specification of Values of Method Parameters

Our solution to specification of the possible values of method parameters is based on the idea that the user defines the set of values which are to be considered as parameters. From the implementation point of view, these sets are to be put into a special Java class serving as a container for all the sets of values. The value of a method parameter of certain type is later nondeterministically selected from the set of values considered for that type and method. In addition to the sets of values common for the whole component, it is also possible to define sets that are specific to a particular method or interface.

Below is a fragment of the specification of values for the Database component:

```java
putIntSet("IDatabase", "insert", new int[]{1, 2, 5, 10});
putIntSet("", "", new int[]{1, 3, 5, 12});
putStringSet("", "", new String[]{"abcd", "EFGH1234"});
```

The first statement defines a set of integer values that is specific to the insert method of the IDatabase interface. The other two statements define the sets of integers and strings that are to be applied to all methods of the Database component interfaces.

The main drawback of this approach is that the user has to define on his/her own the sets of values in such a way that will force the model checker to check all the control flow paths in the component.

5 Evaluation

In this section we compare the two approaches to modeling the environment described above, i.e. the approach of the BEG tool and our approach based on behavior protocols.

The main differences between them are:

- The BEG tool allows to specify parallelism only at the outermost level of regular expressions that specify behavior of the environment (there is no such limit in case of behavior protocols).

- Behavior protocols have no support for method parameters, therefore the possible values of method parameters must be specified separately in a special Java class, while the BEG tool allows to specify the values of method parameters directly in the expressions that specify behavior of the environment.

It is worth to mention that there is also a difference in that the BEG tool targets plain Java classes with informally specified provided and required interfaces, while our approach targets the software components having provided and required interfaces defined in an explicit way.

As a speciality, another advantage of support for specification of parame-
ter values directly in expressions that specify behavior is that it enables the environment generator to select a proper version of an overloaded method - or to generate a code that will non-deterministically invoke all versions of the method that conform to the specification.

We have created an implementation of the environment generator that uses the inverted frame protocol of a component as a model of the environments behavior. It aims at components that use the Fractal Component Model [5] and expects that the Fractal ADL is used to define components. We have successfully applied our environment generator to a component-based application composed of 20 components. Transformations of the frame protocol, described in Sect. 4.3, reduce the size of the state space determined by the protocol approximately thirty times in case of more complex components, therefore lowering also the time required for model checking of the components, all that at the cost of accuracy, though. Nevertheless, model checking of more complex components with environment generated from their frame protocols with no transformations applied is not feasible. Despite the abstractions of the environment introduced by transformations of the frame protocol, the technique is still much more systematic than simple testing. Let us again emphasize that model checking of a component without an environment is not possible at all, because JPF is applicable only to complete Java programs, not isolated software components.

6 Related work

Except for the Bandera Environment Generator [12], we are not aware of any other approach to specification and generation of environment for model checking of software components or parts of object-oriented programs. Nevertheless, there exist model checkers for object-oriented programs that do not need to generate an environment because these tools usually extract a finite model from a complete program (featuring the main method) and then check the model - an example of such a model checker is Zing [2].

There are also tools that solve the problem of automatic generation of environment for fragments of procedural programs (e.g. drivers, libraries, etc). An example of such a tool is the SLAM [4] model checker, which is a part of the SDV tool for verification of device drivers for the Windows operating system. Given a program, the checker creates a Boolean abstraction of the program (all value types approximated by Boolean) and then checks whether some desired temporal properties hold for the abstraction. It uses the principle of refinement to discard errors that are present in the abstraction but not in the original program (false negatives). The environment for device drivers is defined by the interfaces provided by the Windows kernel. The SLAM tool models the environment via training [3]. Here, the basic principle is that, for a certain procedure P that is to be modeled, it first takes several drivers that use the procedure P, then it runs the SDV tool on those drivers and therefore
Parizek, Plasil

gets several Boolean abstractions of the procedure P, and finally merges all those abstractions and puts the resulting Boolean abstraction of the kernel procedure P into a library for future reuse.

Our tool for environment generation is partially based on [10]. The tool that is described in the thesis, designed for the Bandera tool set, also uses the inverted frame protocol idea; it is also focused on components compliant to the Fractal Component Model [5]. We decided not to use this tool mainly because it generates an environment that increases the state space size quite significantly, since it does not employ any of transformations described in Sect. 4.3 and also does not provide any means for specification of method parameter values - all that makes it almost unusable in practice.

7 Conclusion

Direct model checking of isolated software components is usually not possible because model checkers can handle only complete programs. Therefore, it is necessary to create an environment for each component subject to model checking.

In this paper, we have compared two approaches to generating environment of components, resp. classes - namely the Bandera Environment Generator (BEG) tool [12] in Sect. 3, and our approach that is based on behavior protocols [9] in Sect. 4. Main differences between the two approaches lie in the level of support for parallelism, in support for specification of parameter values, and in the fact that the BEG tool is focused on plain Java classes while our approach targets software components with explicitly defined provided and required interfaces.

As to future work, an automated derivation of sets of values used for non-deterministic choice of method parameters is our current goal. It is motivated by the fact that manual definition of such sets requires the user to carefully capture a way that will let the model checker to check all the control flow paths in a target component. A viable approach to the derivation of possible parameter values could be to use static analysis of Java source code (or byte code).

Acknowledgments

We would like to record a special credit to Jiri Adamek and Nicolas Rivierre for valuable comments and Jan Kofron also for many hints regarding the integration of the protocol checker with JPF.
References


