Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 263 (2010) 67–94
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Model Checking of Component Protocol Conformance – Optimizations by Reducing False Negatives

Andreas Both, Wolf Zimmermann and René Franke [1]

Institute of Computer Science
University of Halle
Halle (Saale), Germany

Abstract

In past years, a number of works considered behavioral protocols of components and discussed approaches for automatically checking of compatibility of protocols (*protocol conformance*) in component-based systems. The approaches are usually model-checking approaches, i. e., a positive answer guarantees protocol conformance for all executions while a negative answer provides example executions that may lead to protocol violations. It turned out that if behavioral abstractions take into account unbounded concurrency and unbounded recursion, the protocol conformance checking problem becomes undecidable. There are two possibilities to overcome this problem: (i) further behavioral abstraction to finite state systems or (ii) a conservative approximation of the protocol conformance checking problem. Both approaches may lead to *spurious counterexamples*, i. e., due to abstractions or approximations the shown execution can never happen. This work considers the second approach and shows a heuristics that reduces the number of spurious counterexamples by cutting off search branches that definitely do not contain real counterexamples.

Keywords: verification, protocol conformance, component-based systems, components, model checking, static analysis, process rewrite systems

1 Introduction

Developing software contains nowadays a big share of reusing previously developed software called components. Often these components were developed a long time ago by different developers, third party companies, in different programming languages, supplied as binary code, or as distributed components (e. g., web services). Thus, there is a big interest in components which can be composed to reliable systems.

Creating reliable stateful components is not easy, because the developer has to prevent and catch every possible error, which might possibly be caused by a different sequence of interactions with this component. This observation resulted in

[1] Email: { andreas.both, wolf.zimmermann, rene.franke } @informatik.uni-halle.de

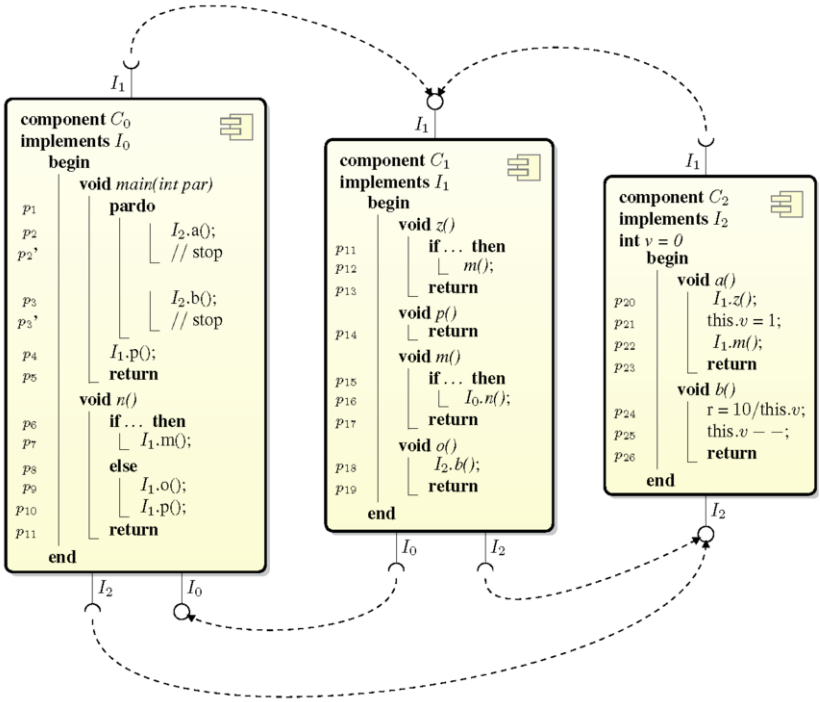
doi:10.1016/j.entcs.2010.05.005

```
interface I0{
  sync void n();
}
```

```
interface I1{
  async void z();
  sync void p();
  sync void m();
  async void o();
}
```

```
interface I2{
  sync void a();
  async void b();
}
```

(a) Interfaces



Calculations influencing only the dataflow and result types are omitted.
Relevant program points are marked with p_i. Synchronous remote method calls are performed blocking, asynchronous remote method calls are performed non-blocking.

(b) Component-based application

$$P_{C_0} = n^* \qquad P_{C_1} = zpm^*(op|po) \qquad P_{C_2} = (a^+b)^*$$

(c) Protocols of the components in Figure 1b

Fig. 1. Example with three interfaces and derived components.

the past years to a number of works on behavioral protocols or interaction protocols. We discuss here a classification of these works. Section 6 contains a more detailed discussion.

In this work, we assume a component model similar to the UML component model. Each component may have provided interfaces (services offered by a component) and required interfaces (services that have to be provided by other components). An interface consists of a set of services which are procedures or functions. A procedure may be implemented either *synchronously* or *asynchronously*. Calling a synchronous procedure blocks the caller until the callee has completed the execution of the procedure. If an asynchronous procedure has been called, a new thread is started and the caller and the callee may proceed their execution concurrently. Fig. 1(a) and (b) show an example.

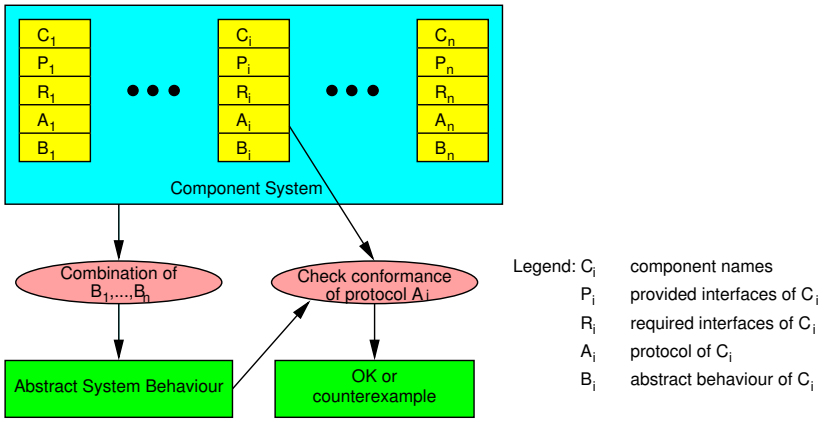


Fig. 2. Automatic Protocol Conformance Checking

There are two basic approaches towards behavioral protocols: (i) The first approach describes a model of the abstract behavior of components specifying how calls to provided interfaces of a component are transformed into calls of required interfaces. Often the abstract behavior is specified as a process-algebraic expression in CSP or CCS (see e. g., [33,31]). Compatibility is checked based on this specifications. (ii) The second approach specifies in addition to the component behavior a protocol for each stateful component (see e. g., [30,21]). The protocol specifies legal sequences of service calls to the provided interfaces of the component. The behavior of the components are used to determine the set of possible sequences of service calls to a particular component. Both approaches are well-suited for automatic protocol conformance checking, since many approaches are restriced to a finite number of states. In this work, we consider the second approach as it clearly specifies the responsibilities of component developers and component users: The component users must obey the protocol while the developers must ensure some desirable properties (such as that component does not abort if the protocol is obeyed). Fig. 1(c) shows an example where the protocols are specified as regular expressions. In the example the protocol of C_2 guarantees that the division at p_{16} never fails. If it is called with b first, it crashes with division by zero.

Fig. 2 summarizes the basic approach of automatic protocol checking. Each component is deployed or published, respectively, with a protocol (to be provided by the component designer) specifying legal sequences of service calls to its provided interfaces and an (automatically derived) abstract component behavior specifying the transitions from calls to provided interfaces to calls to the required interface of a component. In a component system, the abstract component behaviors are composed to an abstract system behavior according to the architecture of the component system. This system behavior can be used to check the conformance to the protocols of each component in the component system. After deployment or publishing a component, we assume that the signatures of provided and required interfaces, the protocol, and the abstract behavior is the only information available for a component. In particular, source code or binary code might be unavailable for component users. Thus, the approach can also deal e. g., with web services. If a

protocol violation is discovered, an example demonstrating the violation is provided (*counterexample*). Since the abstract behavior of a component (and therefore the system behavior) is an abstraction, it may model execution paths that can never be executed. In this case the counterexample is called a *false negative*. Positive answers guarantee that no protocol violations occur.

There are two basic approaches to ensure abstract component behavior. In the *top-down approach*, the component developers specify the abstract behavior and then refine it (e. g., [33,31]). The *bottom-up approach* determines from the source code the abstract behavior (e. g., [43,7]). The top-down approach has the advantage that protocol conformance can be checked before implementing the components while the bottom-up approach has the advantage that the approach can be mechanized using standard control-flow analysis. This work considers the bottom-up approach.

The power of the protocol conformance checking approaches depends on the modeling technique for the behavior and the protocols of components. While component protocols are frequently modeled as finite state machines (see e. g., [30]), the behavior may be modeled as finite state transducers (e. g., [21]), pushdown systems (e. g., [43]), Petri nets (e. g., [35]), process-algebraic expressions (e. g., [33]), etc. The finite state approaches can only model bounded recursion (i. e., up to a certain recursion depth) and bounded concurrency (e. g., by using shuffle automata). Petri nets are able to model unbounded parallelism but are unable to deal with recursion. Pushdown systems can model unbound recursion but cannot model concurrency. Unbounded recursion and unbounded concurrency can be adequately modeled by using process rewrite systems [7,39]. Unfortunately, the protocol conformance checking problem becomes undecidable since LTL model checking is undecidable for process rewrite systems [29,7]. However, reachability checking is still decidable for process rewrite systems.

There are three possibilities to overcome the undecidability of the general protocol conformance checking: (i) further abstraction to, for example finite state systems, (ii) bounded checking (by bounding the recursion depth and the number of concurrent threads), (iii) approximation of the protocol conformance checking. The first and third possibility may lead to more false negatives. The second approach may lead to false positive answers (in case a protocol violation happens with a larger recursion depth or a larger degree of concurrency than the given recursion depth and degree of concurrency, respectively). As demonstrated in [43], a naive implementation of (i) may lead to false positive answers in the presence of recursive call-backs even if each component internally only contains non-recursive procedures. Hence, this work considers the third possibility.

An approximation of the protocol checking problem should be conservative, i. e., if the approximation doesn't discover a protocol violation, then there is indeed no protocol violation. However, an example of a protocol violation may be false because of the approximation. With the information of the abstract system behavior and the protocol, it could be detected whether such a counterexample is a real one (at least w. r. t. the abstract system behavior) or not. For the latter case, one might

refine the approximation of the protocol checking problem that excludes this counterexample and continue to search for further counterexample. This a posteriori approach is similar to a CEGAR-loop in software model checking. This work considers as an alternative an a priori approach: during the search for counterexample a heuristic cuts off branches in the search tree that do not exclude real negatives w.r.t. abstract system behaviors (the branches may contain real negatives but then these are contained in other branches as well).

The paper is organized as follows. Section 2 introduces how process rewrite systems can be used for abstractions. Section 3 introduces the approach of approximate protocol conformance checking and analyzes reasons for spurious counterexamples, while Section 4 shows how to avoid these spurious counterexamples by using a specialized search strategy. Section 5 shows the results of a case study. The paper finishes with a consideration of the related work in Section 6 and the conclusion and future work in Section 7.

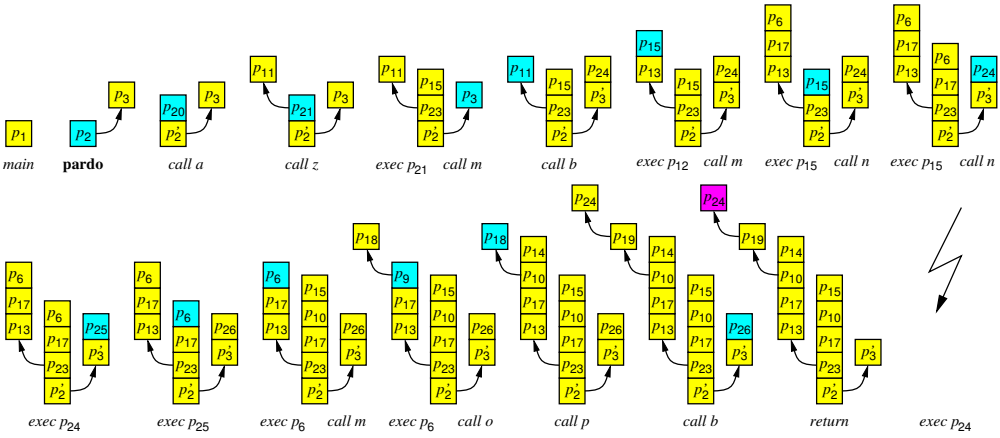
2 Modelling Recursion and Concurrency

This section is a summary of the results in [7] extended by an explanation why the techniques work well. In particular, we present an abstract execution model based on process rewrite systems [29] for concurrent execution of potentially recursive programs and motivate their use. Then, it is shown how abstract component behaviors can be modeled as process rewrite systems and how these are glued together to obtain an abstract system behavior. Finally, we show that special cases lead to component models known from literature.

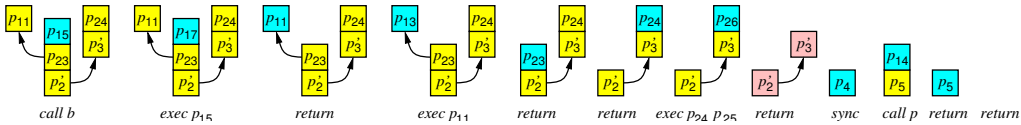
It is well-known that states of imperative programming languages contain a stack to represent the current procedure call hierarchy. A stack is only necessary if the programming language allows recursive procedures. Usually the number of stack frames is infinite. If these are abstracted to a finite number, the abstract model becomes a pushdown automaton (if recursive procedures are allowed) or to finite state machines (if recursive procedures are forbidden). These approaches are used for software model checking since reachability, LTL and CTL model checking is decidable – even for pushdown automata, see for example [13,14].

The natural execution model for capturing unbounded recursion and unbounded concurrency uses states that are represented as a cactus stack (tree of stacks). This model was introduced by [17]. Fig. 3 shows a sample execution of a component system. The stack frames contain the program points to the statements to be executed next. The program points p'_2 and p'_3 denote the program points in *main* after the calls a and b but before their synchronization, respectively. If a procedure call in a process is executed, a stack frame is pushed on a stack. If a new parallel process is started, a new stack grows for this process (like in a saguaro stack). Synchronizations only can be performed between two top-level elements on cactus stacks if one of these stacks contains only one stack frame. Hence, an execution transforms a cactus stack into a cactus stack.

There is a natural correspondence between cactus stacks and process-algebraic



(a) Crashing Computation



(b) Terminating Computation

Fig. 3. Sample Execution of the Component System in Fig. 1

$$
\begin{aligned}
p_1 &\overset{\lambda}{\Rightarrow} p_3 \,||\, p_2 \overset{a}{\Rightarrow} p_3 \,||\, p_{20}.p_2' \overset{z}{\Rightarrow} p_3 \,||\, (p_{11} \,||\, p_{21}).p_2' \overset{\lambda}{\Rightarrow} p_3 \,||\, (p_{11} \,||\, p_{22}).p_2' \overset{m}{\Rightarrow} p_3 \,||\, (p_{11} \,||\, p_{15}.p_{23}).p_2' \\
&\overset{b}{\Rightarrow} p_{24}.p_3' \,||\, (p_{11} \,||\, p_{15}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_{12} \,||\, p_{15}.p_{23}).p_2' \overset{m}{\Rightarrow} p_{24}.p_3' \,||\, (p_{15}.p_{13} \,||\, p_{15}.p_{23}).p_2' \\
&\overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_{16}.p_{13} \,||\, p_{15}.p_{23}).p_2' \overset{n}{\Rightarrow} p_{24}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_{15}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_{16}.p_{23}).p_2' \\
&\overset{n}{\Rightarrow} p_{24}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_6.p_{17}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{25}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_6.p_{17}.p_{23}).p_2' \\
&\overset{\lambda}{\Rightarrow} p_{26}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_6.p_{17}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{26}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_7.p_{17}.p_{23}).p_2' \\
&\overset{m}{\Rightarrow} p_{26}.p_3' \,||\, (p_6.p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{26}.p_3' \,||\, (p_8.p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2' \\
&\overset{o}{\Rightarrow} p_{26}.p_3' \,||\, ((p_{18} \,||\, p_9).p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2' \overset{p}{\Rightarrow} p_{26}.p_3' \,||\, ((p_{18} \,||\, p_{14}.p_{10}).p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2' \\
&\overset{b}{\Rightarrow} p_{26}.p_3' \,||\, (p_{24} \,||\, p_{19} \,||\, p_{14}.p_{10}).p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2' \\
&\overset{\lambda}{\Rightarrow} p_3' \,||\, (p_{24} \,||\, p_{19} \,||\, p_{14}.p_{10}).p_{17}.p_{13} \,||\, p_{15}.p_{10}.p_{17}.p_{23}).p_2'
\end{aligned}
$$

(a) Crashing Computation (cf. Figure 3a)

$$
\begin{aligned}
&\overset{b}{\Rightarrow} p_{24}.p_3' \,||\, (p_{11} \,||\, p_{15}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_{11} \,||\, p_{17}.p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_{11} \,||\, p_{23}).p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (p_{13} \,||\, p_{23}).p_2' \\
&\overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, (\varepsilon \,||\, p_{23}).p_2' = p_{24}.p_3' \,||\, p_{23}.p_2' \overset{\lambda}{\Rightarrow} p_{24}.p_3' \,||\, p_2' \overset{\lambda}{\Rightarrow} p_{25}.p_3' \,||\, p_2' \overset{\lambda}{\Rightarrow} p_{26}.p_3' \,||\, p_2' \overset{\lambda}{\Rightarrow} p_3' \,||\, p_2' \overset{\lambda}{\Rightarrow} p_4 \overset{p}{\Rightarrow} p_{14}.p_5 \\
&\overset{\lambda}{\Rightarrow} p_5 \overset{\lambda}{\Rightarrow} \varepsilon
\end{aligned}
$$

(b) Terminating Computation (cf. Figure 3b)

Fig. 4. Sample Execution as Process Rewrites corresponding to the Executions in Fig. 3

expressions. Let Q be a finite set of atomic processes (in our case the set of program points). A *process-algebraic expression over* Q is inductively defined as follows:

i. Any $q \in Q$ is a process-algebraic expression and the *empty process* ε is a process-algebraic expression (where $\varepsilon \notin Q$).

ii. If q_1 and q_2 are process-algebraic expressions, then $q_1 \,||\, q_2$ (*parallel composition*) and $q_1.q_2$ (*sequential composition*) are process-algebraic expressions.

The empty process ε is the identity w. r. t. the binary operators . and $||$. Both binary operators are associative and the operator $||$ is commutative. In order to avoid parentheses we assume that the operator . has higher priority than $||$. A stack in a cactus stack is represented by the sequential composition. The branching in a cactus

stack is represented by parallel composition. Fig. 4 shows the process-algebraic expressions representing the cactus stacks in Fig. 3. The derivation $q_1 \stackrel{a}{\Rightarrow} q_2$ denotes that the cactus stack q_2 succeeds q_1 and procedure a was called and $q_1 \stackrel{\lambda}{\Rightarrow} q_2$ that no procedure call happened. Note that not all process-algebraic expressions correspond to cactus stacks. For example an expression $p_1.(p_2 \,||\, p_3)$ cannot correspond to a cactus stack because $p_2 \,||\, p_3$ would represent two branches in a cactus stack and therefore it would not be clear where to put p_1 in the cactus stack.

If only program points are considered, then the abstract states correspond to process-algebraic expressions over the program points and an abstract execution is a sequence of process-algebraic expressions. An abstract state transformation rewrites a process-algebraic expression into a process-algebraic expression. Process rewrite systems (PRS) [29] are a descriptive technique for such transformations. Let $PEX(Q)$ be the set of process-algebraic expressions over a finite set Q of atomic processes. A *process rewrite system* is a tuple $\Pi \hat{=} (Q, \Sigma, I, \rightarrow, F)$, where Q is a finite set of atomic processes, Σ is a finite alphabet over actions, $I \in Q$ is the initial process, $\rightarrow \subseteq PEX(Q) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q)$ is a set of process rewrite rules, $F \subseteq PEX(Q)$ is a finite set of final processes, and $\lambda \notin \Sigma$ (*no relevant interaction* or *empty word*). The process rewrite rules define a derivation relation $\Rightarrow \in PEX(Q) \times \Sigma^* \times PEX(Q)$ by the following inference rules ($a \in \Sigma, x \in \Sigma^*$):

$$\frac{(t_1 \stackrel{a}{\rightarrow} t_2) \in \Pi}{t_1 \stackrel{a}{\Rightarrow} t_2}, \quad \frac{t_1 \stackrel{a}{\Rightarrow} t_2}{t_1.t_3 \stackrel{a}{\Rightarrow} t_2.t_3}, \quad \frac{t_1 \stackrel{a}{\Rightarrow} t_2}{t_1||t_3 \stackrel{a}{\Rightarrow} t_2||t_3}, \quad \frac{t_1 \stackrel{a}{\Rightarrow} t_2}{t_3||t_1 \stackrel{a}{\Rightarrow} t_3||t_2}, \quad \frac{t_1 \stackrel{x}{\Rightarrow} t_2 \;\; t_2 \stackrel{a}{\Rightarrow} t_3}{t_1 \stackrel{x\,a}{\Longrightarrow} t_3}$$

The set $L(\Pi) \hat{=} \{w : \exists f \in F | I \stackrel{w}{\Rightarrow} f\}$ is the *language accepted* by Π.

In our case Q corresponds to the set of program points and Σ to the set of procedure names in the interfaces. Table 1 shows how a set of rewrite rules can be derived from a program. It assumes that all program points are made explicit [2]. Calling a sequential procedure proceeds and pushes on the stack a new element with the first program point of the procedure. An asynchronous procedure call instead forks a new parallel process. A procedure return simply rewrites a process into the empty process. Furthermore, in our case we only consider $F = \{\varepsilon\}$ since an execution regularly terminates with the empty cactus stack.

A closer look on the derivation rules shows that they formalize exactly the abstract execution model: The second rule for the definition of $\Rightarrow$ formalizes that transformations are applied only to elements on the top of the stacks of a cactus stack. The third and forth rule specify that any stack in a cactus stack can be considered (i. e., each of the processes currently being executed can be selected for state transformations). Thus, these two rules model interleaving semantics.

If all components are considered as white-box components, then the PRS obtained by the transformations in Table 1 yields a PRS defining the abstract system behavior. Fig. 5 shows the rewrite rules of the abstract system behavior of the component-based system in Fig. 1. The derivations in Fig. 4 can be construced with the PRS of Fig. 5.

[2] This is done in every compiler.

$\mathcal{A}[\![s_1; s_2]\!] \triangleq \mathcal{A}[\![s_1]\!] \cup \mathcal{A}[\![s_2]\!]$

$\mathcal{A}[\![p_1 : \textbf{if } \varphi \textbf{ then } p_2 : s]\!] \triangleq \mathcal{A}[\![s]\!] \cup \{p_1 \xrightarrow{\lambda} p_2\} \cup \{p_1 stackrel \lambda \rightarrow p_3 : p_3 \in \mathcal{N}[\![p_1 : \textbf{if } \varphi \textbf{ then } p_2 : s]\!]\}$

$\mathcal{A}[\![p_1 : \textbf{if } \varphi \textbf{ then } p_2 : s_2 \textbf{ else } p_3 : s_3]\!] \triangleq \mathcal{A}[\![s_1]\!] \cup \mathcal{A}[\![s_2]\!] \cup \{p_1 \xrightarrow{\lambda} p_2, p_1 \xrightarrow{\lambda} p_3\}$

$\mathcal{A}[\![p_1 : \textbf{while } \varphi \textbf{ do } p_2 : s]\!] \triangleq \mathcal{A}[\![s]\!] \cup \{p_1 \xrightarrow{\lambda} p_2\} \cup \{p \xrightarrow{\lambda} p_1 : p \in \mathcal{L}[\![s]\!]\} \cup$
$\{p_1 \xrightarrow{\lambda} p : p \in \mathcal{N}[\![p_1 : \textbf{while } \varphi \textbf{do } p_2 : s]\!]\}$

$\mathcal{A}[\![p_1 : \textbf{pardo } p_2 : s_2; p_2' \,||\, p_3 : s_3; p_3']\!] \triangleq \mathcal{A}[\![s_2]\!] \cup \mathcal{A}[\![s_3]\!] \cup \{p_1 \xrightarrow{\lambda} p_2 \,||\, p_3\} \cup$
$\{p_2' \,||\, p_3' \xrightarrow{\lambda} p_4 : p_4 \in \mathcal{N}[\![p_1 : \textbf{pardo } p_2 : s_2; p_2' \,||\, p_3 : s_3; p_3']\!]\}$

$\mathcal{A}[\![p : m()]\!] \triangleq \{p \xrightarrow{m} first(m).p' : p' \in \mathcal{N}[\![p : m()]\!]\}$ if m is synchronous

$\mathcal{A}[\![p : m()]\!] \triangleq \{p \xrightarrow{m} first(m) \,||\, p' : p' \in \mathcal{N}[\![p : m()]\!]\}$ if m is asynchronous

$\mathcal{A}[\![p : \textbf{return}]\!] \triangleq \{p \xrightarrow{\lambda} \varepsilon\}$

$\mathcal{A}[\![p : s]\!] \triangleq \{p \xrightarrow{\lambda} p' : p' \in \mathcal{N}[\![p : s]\!]\}$ if s is atomic and no procedure call or return

$\mathcal{A}[\![\textbf{void } m(\cdots)\ s]\!] \triangleq \mathcal{A}[\![s]\!]$

$\mathcal{A}[\![proc_1; proc_2]\!] \triangleq \mathcal{A}[\![proc_1]\!] \cup \mathcal{A}[\![proc_2]\!]$

Program Points are denoted on the statements

$\mathcal{L}[\![s]\!]$ denotes the set of last program points in a statement sequence s

$\mathcal{N}[\![s]\!]$ denotes the set of program points that follow statement s

$first(m)$ is the first program point of procedure m.

Table 1
Abstraction of Programs to Process Rewrite Systems

$main$	$p_1 \xrightarrow{\lambda} p_3 \,\|\|\, p_2,\ \ p_2 \xrightarrow{a} p_{20}.p_2',\ \ p_3 \xrightarrow{b} p_{24} \,\|\|\, p_3',\ \ p_3' \,\|\| p_2' \xrightarrow{\lambda} p_4,\ \ p_4 \xrightarrow{p} p_{14}.p_5,\ \ p_5 \xrightarrow{\lambda} \varepsilon$
n	$p_6 \xrightarrow{\lambda} p_7,\ p_6 \xrightarrow{\lambda} p_8,\ p_7 \xrightarrow{m} p_{15}.p_{10},\ p_8 \xrightarrow{o} p_{18} \,\|\|\, p_9,\ p_9 \xrightarrow{p} p_{14}.p_{10},\ p_{10} \xrightarrow{\lambda} \varepsilon$
z	$p_{11} \xrightarrow{\lambda} p_{12},\ p_{11} \xrightarrow{\lambda} p_{13},\ p_{12} \xrightarrow{m} p_{15}.p_{13},\ p_{13} \xrightarrow{\lambda} \varepsilon$
p	$p_{14} \xrightarrow{\lambda} \varepsilon$
m	$p_{15} \xrightarrow{\lambda} p_{16},\ p_{15} \xrightarrow{\lambda} p_{17},\ p_{16} \xrightarrow{n} p_6.p_{17},\ p_{17} \xrightarrow{\lambda} \varepsilon$
o	$p_{18} \xrightarrow{b} p_{24} \,\|\|\, p_{19},\ p_{19} \xrightarrow{\lambda} \varepsilon$
a	$p_{20} \xrightarrow{z} p_{11} \,\|\|\, p_{21},\ p_{21} \xrightarrow{\lambda} p_{22},\ p_{22} \xrightarrow{m} p_{15}.p_{22},\ p_{23} \xrightarrow{\lambda} \varepsilon$
b	$p_{24} \xrightarrow{\lambda} p_{25},\ p_{25} \xrightarrow{\lambda} p_{26},\ p_{26} \xrightarrow{\lambda} \varepsilon$

Fig. 5. Abstract System Behavior (as a Process Rewrite System) of the Component System in Fig. 1

Remark: As it can be seen from Table 1, the process rewrite rules contain on its LHS at most the parallel operator and the RHS is a process-algebraic expression with at most either a parallel operator or a sequential operator. In [29] this restriction is called Process Algebra Nets (PAN). It can handle unbounded recursion and unbounded parallelism including synchronization. This work uses PAN as representation of the abstract system behavior (and as the abstract behavior of components). Mayr shows in [29] that the rules of any PRS can be transformed into a *normal form*, i. e., the LHS and the RHS has one of the forms q_1, $q_1.q_2$ or $q_1||q_2$, where q_1 and q_2 are atomic processes. Thus, we implicitly assume that the PRS are in normal form. ∎

Remark: Finite state automata are represented by the special class of PRS where the process rewrite rules only rewrite atomic processes into atomic processes. Pushdown automata are represented by the class of PRS, which allow no parallel operator. The special case where the LHS of a process rewrite rule consists only of an atomic process is suffcient. In contrast Petri nets are represented by the class of PRS, which allow no sequential operator. Thus, PRS allowing the use of both operators unify the behavior of pushdown automata and Petri nets. ∎

An abstract component behavior basically can be derived from its implementation analogously to the abstract system behavior. The only difference is the treat-

$$
\begin{array}{lll}
C_0: & main & p_1 \xrightarrow{\lambda} p_3 \,||\, p_2,\ p_2 \xrightarrow{a} q_{I_2,a}.p_2',\ p_3 \xrightarrow{b} q_{I_2,b} \,||\, p_3', p_3' \,||\, p_2' \xrightarrow{\lambda} p_4,\ p_4 \xrightarrow{p} q_{I_1,p}.p_5,\ p_5 \xrightarrow{\lambda} \varepsilon \\
 & n & p_6 \xrightarrow{\lambda} p_7,\ p_6 \xrightarrow{\lambda} p_8,\ p_7 \xrightarrow{m} q_{I_1,m}.p_{10},\ p_8 \xrightarrow{o} q_{I_1,o} \,||\, p_9,\ p_9 \xrightarrow{p} q_{I_1,p}.p_{10},\ p_{10} \xrightarrow{\lambda} \varepsilon \\
C_1: & z & p_{11} \xrightarrow{\lambda} p_{12},\ p_{11} \xrightarrow{\lambda} p_{13},\ p_{12} \xrightarrow{m} q_{I_1,m}.p_{13},\ p_{13} \xrightarrow{\lambda} \varepsilon \\
 & p & p_{14} \xrightarrow{\lambda} \varepsilon \\
 & m & p_{15} \xrightarrow{\lambda} p_{16},\ p_{15} \xrightarrow{\lambda} p_{17},\ p_{16} \xrightarrow{n} q_{I_0,n}.p_{17},\ p_{17} \xrightarrow{\lambda} \varepsilon \\
 & o & p_{18} \xrightarrow{b} q_{I_2,b} \,||\, p_{19},\ p_{19} \xrightarrow{\lambda} \varepsilon \\
C_2: & a & p_{20} \xrightarrow{\lambda} q_{I_1,z} \,||\, p_{21},\ p_{21} \xrightarrow{\lambda} p_{22},\ p_{22} \xrightarrow{m} q_{I_1,m}.p_{22},\ p_{23} \xrightarrow{\lambda} \varepsilon \\
 & b & p_{24} \xrightarrow{\lambda} p_{25},\ p_{25} \xrightarrow{\lambda} p_{26},\ p_{26} \xrightarrow{\lambda} \varepsilon
\end{array}
$$

Fig. 6. Abstract System Behavior (as a Process Rewrite System) of the Component System in Fig. 1

ment of calls of services of the component's required interfaces. Before connecting a required interface of a component C to the provided interface of another component C', there is no known program point. For an isolated consideration of a component C, we use a dummy program point $q_{I,m}$ for each service m of a required interface I of C. After composition with a provided interface I of a component C', $q_{I,m}$ is replaced by the first program point $first_{C'}(m)$ of component C'. Thus, the abstract behavior of a component C can be modeled as a tuple $B_C \triangleq (\mathbb{P}, \mathbb{R}, Q_C, first_C, \rightarrow)$ where $\mathbb{P}$ and $\mathbb{R}$ is the provided interface of C, an $\mathcal{R}$ is the set of the required interface of C, $first_C : \mathbb{P} \rightarrow Q_C$ is a mapping specifying the first program point in C for the services $m \in \mathbb{B}$, and $\rightarrow$ is a finite set of process rewrite rules over the alphabet $\mathbb{P} \cup \bigcup_{I \in \mathbb{R}} I$ and the atomic processes $Q \cup \{q_{I,m} : I \in \mathbb{R}, m \in I\}$. Fig. 6 shows the abstract component behaviors of the components in Fig. 1.

The process rewrite rules are similar to the process rewrite rules in Fig. 5: if $q_{I_0,n}$ is replaced by $p_6 = first_{C_0}(n)$, $q_{I_1,z}$ is replaced by $p_{11} = first_{C_1}(z)$, $q_{I_1,p}$ is replaced by $p_{14} = first_{C_1}(p)$, $q_{I_1,m}$ is replaced by $p_{15} = first_{C_1}(m)$, $q_{I_1,o}$ is replaced by $p_{18} = first_{C_1}(o)$, $q_{I_2,a}$ is replaced by $p_{20} = first_{C_2}(a)$, and $q_{I_2,b}$ is replaced by $p_{24} = first_{C_2}(b)$, then the abstract system behavior in Fig. 5 is obtained. The above replacements are defined according to the architecture in Fig. 1: each $q_{I,m}$ in the abstract behavior of component C is replaced by $first_{C'}(m)$ if the required interface I of C is bounded the provided interface I of C'. Thus, if the deployed components publish their abstract component behavior, all information is available for obtaining the abstract system behavior of a component system as a process rewrite system.

Thus, abstract component behaviors can be modeled by a set of process-rewrite rules. It is possible to automatically derive them from the source code of the components. For a component-based system S, the abstract behaviors of its components can be composed according to the architecture in order to obtain the abstract system behavior of S. This step only requires the abstract behavior of the components but does not require their implementation. Since finite state machines, pushdown machines, and Petri nets are special cases of process rewrite systems, previous approaches based on finite transducers, pushdown systems or Petri nets can be modeled as well as with process rewrite systems.

In [29] it is shown that reachability problems (and therefore the word problem) are decidable for general PRS, that LTL-model checking is decidable for PRS that either only use sequential operators or only use parallel operators, and that LTL-model checking becomes undecidable for PRS that use parallel operators as well as sequential operators. The restrictions for decidability correspond to push-

down systems (only sequential operator) and Petri nets (only parallel operator), respectively.

3 Protocol Conformance Checking

A *component protocol* (short: protocol) describes the allowed use of all callable operations (cf. interfaces) of a single component. It specifies the allowed sequences of incoming invocations to the component and therefore represents a contract based only on the interface description. Among other scopes of applications, protocols can be used to avoid uncaught exceptions (cf. [7]) during the execution of a component or to obey business rules (cf. [6,8]) For example, an SSO-component [3] may offer the following services: register and sign in (service a), sign in (b), optionally change password (c), logout (d). The component could have the following protocol (business rule) formulated as regular expression: $((a|b)c^*d)^*$. This protocol should be obeyed by every caller of the component.

Protocols can be verified dynamically by performing the transitions according to the (incoming remote) invocations. In this work we focus on static verification of protocols, i. e., it is verified that a component is used according to its protocol before executing the component-based system.

In accordance with other works [43,36,20] protocols are defined by finite state machines (short: FSM): the protocol P_C of a component C is a FSM $P_C \hat{=} (Q_{P_C}, \Sigma_{P_C}, \rightarrow_{P_C}, I_{P_C}, F_{P_C})$, where Q_{P_C} is a finite set of states, Σ_{P_C} is a finite set of services, $\rightarrow_{P_C} \subseteq Q_{P_C} \times \Sigma_{P_C} \times Q_{P_C}$ is a finite set of transition rules, $I_{P_C} \in Q_{P_C}$ is the initial state, $F_{P_C} \subseteq Q_{P_C}$ is the set of final states. Figure 1 shows an example of a component system implementing different interfaces. The protocols of the components are shown as regular expressions (which are equal to FSMs). We will consider only the protocol of the component C_2 [4]. This component might crash on a division by zero exception if it is called with the sequence $a\,b\,b$, for example. If the protocol $(a^+b)^*$ is obeyed, this crash can be avoided.

Let Π_S be the abstract behavior of a component system S (specified as a PRS) and C be a component of S with protocol P_C. The calls to component C can be obtained from the abstract system behavior Π_S by replacing all actions by λ except for the services of the provided interface of C. This PRS $\Pi_{S,C}$ is called the *use of component* C. Hence, the protocol conformance checking is equivalent to check whether $L(\Pi_{S,C}) \subseteq L(P_C)$, where $L(P_C)$ is a regular language defined by the protocol P_C of the component C. Table 2 summarizes the notations in this paper. Fig. 7 shows the use of component C_2 for the example in Fig. 1. Fig. 7(b) shows that *abb* is a use that violates the protocol of C_2. This derivation corresponds to the sample execution in Fig. 3(a) and Fig. 4(a), respectively.

The undecidability of LTL-model-checking under PRS that use both parallel and sequential operators [29] implies that protocol conformance checking becomes

[3] Single Sign On. A component which provides the functionality of a login/logout/session management, so different applications can use this mechanism to verify a user.

[4] All other protocols can be checked using the similar way.

$$
\begin{array}{ll}
main & p_1 \xrightarrow{\lambda} p_3 \,||\, p_2,\ p_2 \xrightarrow{a} p_{20}.p_2',\ p_3 \xrightarrow{b} p_{24} \,||\, p_3', p_3' \,||\, p_2' \xrightarrow{\lambda} p_4,\ p_4 \xrightarrow{\lambda} p_{14}.p_5,\ p_5 \xrightarrow{\lambda} \varepsilon \\
n & p_6 \xrightarrow{\lambda} p_7,\ p_6 \xrightarrow{\lambda} p_8,\ p_7 \xrightarrow{\lambda} p_{15}.p_{10},\ p_8 \xrightarrow{\lambda} p_{18} \,||\, p_9,\ p_9 \xrightarrow{\lambda} p_{14}.p_{10},\ p_{10} \xrightarrow{\lambda} \varepsilon \\
z & p_{11} \xrightarrow{\lambda} p_{12},\ p_{11} \xrightarrow{\lambda} p_{13},\ p_{12} \xrightarrow{\lambda} p_{15}.p_{13},\ p_{13} \xrightarrow{\lambda} \varepsilon \\
p & p_{14} \xrightarrow{\lambda} \varepsilon \\
m & p_{15} \xrightarrow{\lambda} p_{16},\ p_{15} \xrightarrow{\lambda} p_{17},\ p_{16} \xrightarrow{\lambda} p_6.p_{17},\ p_{17} \xrightarrow{\lambda} \varepsilon \\
o & p_{18} \xrightarrow{b} p_{24} \,||\, p_{19},\ p_{19} \xrightarrow{\lambda} \varepsilon \\
a & p_{20} \xrightarrow{\lambda} p_{11} \,||\, p_{21},\ p_{21} \xrightarrow{\lambda} p_{22},\ p_{22} \xrightarrow{\lambda} p_{15}.p_{22},\ p_{23} \xrightarrow{\lambda} \varepsilon \\
b & p_{24} \xrightarrow{\lambda} p_{25},\ p_{25} \xrightarrow{\lambda} p_{26},\ p_{26} \xrightarrow{\lambda} \varepsilon
\end{array}
$$

Fig. 7. Use of Component C_2 in the Example of Fig. 1

Notation	Description
P_C	protocol of a component C (represented as FSM or regular expression)
Π_S	abstract system behavior of a full component-based system
$\Pi_{S,C}$	abstract system behavior, considering only the interactions of component C
Π_S^C	Combined Abstraction of a component C
$PEX(Q)$	set of process-algebraic expressions computable using Q
s, t	process-algebraic terms, $s, t \in PEX(Q)$
v	states of an (inverted) protocol, $v \in P_C$
a, b, λ	interactions in the considered system, $a, b \in \Sigma$, no interaction $\lambda \notin \Sigma$
w, x, y	sequences of interactions, $w, x, y \in \Sigma^*$
λ	empty sequence of interactions
p, ε	atomic processes ($p \in Q$), empty process ($\varepsilon \in Q$)

Table 2
Notations used in this paper.

undecidable [7], provided that the use of a component contains parallel as well as sequential operators. However, reachability remains decidable [29]. There are the following alternatives to tackle the problem of undecidability of protocol conformance checking: (i) *Bounded protocol conformance checking* bounds recursion depth and the degree of parallelism, (ii) further approximation of the abstract component behavior and abstract system behavior, and (iii) approximation of the protocol conformance checking problem by a reachability problem. The first alternative is often used as an argument for using finite state systems for modeling abstract component behaviors. Although it is likely to discover protocol violations with a bounded approach, false positives are not excluded because a protocol may be violated by using a larger recursion depth or a larger degree of parallelism than the given bounds. At first glance, the second approach seems promising. However, [43] shows an example of a (sequential) component system with two components, both having a finite state transducer as an abstract behavior, but protocol conformance checking leads to false positives due to a recursive call-back. Furthermore, it is more likely to discover false negatives that cannot be discovered without component implementations. We therefore consider the third approach because this allows at least to check on the level of the abstract system behavior whether a counterexample is real or spurious.

The basic idea to approximate the protocol conformance checking is to construct a PRS Π_S^C such that $L(\Pi_S^C) \supseteq \overline{L(P_C)} \cap L(\Pi_{S,C})$. Hence, Π_S^C contains all sequences

$$
\begin{aligned}
\mathcal{R}^1_C &= \{\ (v,p,v'') \xrightarrow{a} (v',p',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p') && \}\\
\mathcal{R}^\lambda_C &= \{\ (v,p,v') \xrightarrow{\lambda} (v,p',v') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p') && \}\\
\mathcal{R}^1_{Seq} &= \{\ (v,p,v''') \xrightarrow{a} (v',p',v'').(v'',p'',v''') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'.p'') && \}\\
\mathcal{R}^\lambda_{Seq} &= \{\ (v,p,v'') \xrightarrow{\lambda} (v,p',v').(v',p'',v'') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'.p'') && \}\\
\mathcal{R}^1_{PFork} &= \{\ (v,p,v'') \xrightarrow{a} (v',p',v'')||(v',p'',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'||p'') && \}\\
\mathcal{R}^\lambda_{PFork} &= \{\ (v,p,v') \xrightarrow{\lambda} (v,p',v')||(v,p'',v') : && (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'||p'') && \}\\
\mathcal{R}^1_{PSync} &= \{\ (v,p,v'')||(v,p',v'') \xrightarrow{a} (v',p'',v'') : && (v \xrightarrow{a}_{P_C} v') \wedge (p||p' \xrightarrow{a}_{\Pi_{S,C}} p'') && \}\\
\mathcal{R}^\lambda_{PSync} &= \{\ (v,p,v')||(v,p',v') \xrightarrow{\lambda} (v,p'',v') : && (p||p' \xrightarrow{\lambda}_{\Pi_{S,C}} p'') && \}\\
\mathcal{R}^0 &= \{\ (v,p,v'') \xrightarrow{\lambda} (v',p,v'') : && (v \xrightarrow{a}_{P_C} v') && \}\\
\mathcal{R}^\varepsilon &= \{\ (v,\varepsilon,v) \xrightarrow{\lambda} \varepsilon && && \}\\
&\qquad \text{with} \quad v,v',v'',v''' \in Q_A; \quad p,p',p'' \in Q_S; \quad a \in \Sigma_{P_C}
\end{aligned}
$$

Fig. 8. Rules for creating the Combined Abstraction Π^C_S.

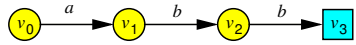
of interactions that are forbidden [5] by the protocol P_C of one component C (i. e., $\overline{P_C}$) and exists in the program S. Since reachability is decidable for PRS it can be decided whether $L(\Pi^C_S) = \emptyset$. Thus, $L(\Pi^C_S) = \emptyset$ implies that $L(\Pi_{S,C}) \subseteq L(P_C)$. Furthermore, a counterexample $w \in L(P_C)$ is spurious iff $w \notin \overline{L(P_C)} \cap L(\Pi_{S,C})$. Since the word problem is a specialized reachability problem, it can be decided whether a counterexample is spurious.

The transition rules $\rightarrow_{\Pi^C_S} = \mathcal{R}^1_C \cup \mathcal{R}^\lambda_C \cup \mathcal{R}^1_{Seq} \cup \mathcal{R}^\lambda_{Seq} \cup \mathcal{R}^1_{PFork} \cup \mathcal{R}^\lambda_{PFork} \cup \mathcal{R}^1_{PSync} \cup \mathcal{R}^\lambda_{PSync} \cup \mathcal{R}^0 \cup \mathcal{R}^\varepsilon$ of the Combined Abstraction Π^C_S are computed by using the directives in Fig. 8. All process constants are triples (v_i, p_k, v_j) where $v_i, v_j \in Q_{P_C}$ and $p_k \in Q_S$. A triple (v_i, p_k, v_j) encodes the situation, where p_k should be rewritten to the empty process $p_k \overset{w}{\Rightarrow}_{\Pi_S} \varepsilon$, while the state v_i of the FSM P_C is transformed into v_j accepting the same word w: $v_i \overset{w}{\Rightarrow}_{P_C} v_j$. A constant (v_j, ε, v_j) is equivalent to the empty word, because the targeted protocol state is reached and process constants have been eliminated (cf. $\mathcal{R}^\varepsilon$ in Fig. 8). Technical details of the construction of the Combined Abstraction Π^C_S can be found in [7]. The *Combined Abstraction* Π^C_S is a PRS in the same class of the PRS-hierarchy (cf. [29]) as Π_S.

The construction of the Combined Abstraction is similar to the intersection of a language accepted by a PDA with a language accepted by a FSM (cf. [23]): this yields a PDA consisting of the rules $\mathcal{R}^1_C \cup \mathcal{R}^\lambda_C \cup \mathcal{R}^1_{Seq} \cup \mathcal{R}^\lambda_{Seq} \cup \mathcal{R}^\varepsilon$. The rules $\mathcal{R}^1_{PFork} \cup \mathcal{R}^\lambda_{PFork} \cup \mathcal{R}^1_{PSync} \cup \mathcal{R}^\lambda_{PSync}$ are straightforward generalizations of the approach in [23]. The rules $\mathcal{R}^0$ require a further explanation. Consider the example in Fig. 9(a). Fig. 9(b) shows the Combined Abstraction. The sets $\mathcal{R}^1_C$, $\mathcal{R}^\lambda_{Seq}$, $\mathcal{R}^\lambda_{PFork}$, $\mathcal{R}^1_{PSync}$, and $\mathcal{R}^\lambda_{PSync}$ are all empty. In order to check whether the undesired state v_3 is reached, it is sufficient to check whether there is a derivation $(v_0, p_0, v_3) \overset{w}{\Rightarrow} \varepsilon$ in the Combined Abstraction. Fig. 9(c) shows that there is no such derivation without using $\mathcal{R}^0$-rules. There is no rule except $\mathcal{R}^0$-rules in the Combined Abstraction that can be applied to the last process-algebraic expressions, respectively. Thus, these two derivations cannot be continued. The derivations $\mathfrak{c}'_0$ and $\mathfrak{c}'_1$ are in some sense representative, since each

[5] These sequences are represented by the inverted protocol, i. e., the inverted FSM $\overline{P_C}$ accepting the complement $\overline{L(P_C)} \hat{=} \Sigma^* \setminus L(P_C)$.

$p_0 \xrightarrow{a} p_2||p_1,\ p_1 \xrightarrow{b} p_5.p_3,\ p_2 \xrightarrow{\lambda} \varepsilon,\ p_2 \xrightarrow{b} p_5.p_4,\ p_2 \xrightarrow{b} p_5.p_2,\ p_3 \xrightarrow{\lambda} \varepsilon,\ p_4 \xrightarrow{\lambda} \varepsilon,\ p_5 \xrightarrow{\lambda} \varepsilon$



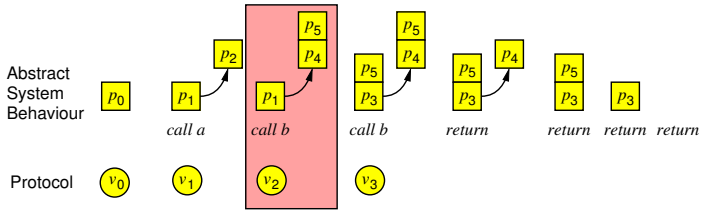
(a) Abstract System Behavior and (inverted) Protocol with forbidden interaction sequence

$$\begin{aligned}
\mathcal{R}^{\lambda}_{C} &= \{(v_i,p_3,v_j) \xrightarrow{\lambda} (v_i,\varepsilon,v_j),(v_i,p_4,v_j) \xrightarrow{\lambda} (v_i,\varepsilon,v_j),(v_i,p_5,v_j) \xrightarrow{\lambda} (v_i,\varepsilon,v_j) : 0 \le i,j \le 3,\}\\
\mathcal{R}^{1}_{Seq} &= \{(v_1,p_1,v_j) \xrightarrow{b} (v_2,p_5,v_k).(v_k,p_3,v_j),(v_1,p_2,v_j) \xrightarrow{b} (v_2,p_5,v_k).(v_k,p_4,v_j) : 0 \le j,k \le 3\}\\
\mathcal{R}^{1}_{PFork} &= \{(v_0,p_0,v_j) \xrightarrow{a} (v_1,p_1,v_j)||(v_1,p_2,v_j) : 0 \le j \le 3\}\\
\mathcal{R}^{\varepsilon} &= \{(v_i,\varepsilon,v_i) \xrightarrow{\lambda} \varepsilon : 0 \le i \le 3\}\\
\mathcal{R}^{0} &= \{(v_0,p_i,v_j) \xrightarrow{\lambda} (v_1,p_i,v_j),\ (v_1,p_i,v_j) \xrightarrow{\lambda} (v_2,p_i,v_j),\ (v_2,p_i,v_j) \xrightarrow{\lambda} (v_3,p_i,v_j) : 0 \le i \le 5, 0 \le j \le 3\}
\end{aligned}$$

(b) Combined Abstraction

$$\begin{aligned}
\mathfrak{c}'_0 :\ (v_0,p_0,v_3) &\xRightarrow{a} (v_1,p_2,v_3)||(v_1,p_2,v_3) \xRightarrow{b} (v_1,p_2,v_3)||(v_2,p_5,v_2).(v_2,p_4,v_3)\\
&\xRightarrow{\lambda} \underline{(v_1,p_2,v_3)||(v_2,\varepsilon,v_2).(v_2,p_4,v_3)} \xRightarrow{\lambda} (v_1,p_2,v_3)||(v_2,p_4,v_3) \xRightarrow{\lambda} (v_1,p_2,v_3)||(v_2,\varepsilon,v_3)\\
&\xRightarrow{b} \underline{(v_2,p_5,v_2).(v_2,p_3,v_3)||(v_2,\varepsilon,v_3)} \xRightarrow{\lambda} (v_2,\varepsilon,v_2).(v_2,p_3,v_3)||(v_2,\varepsilon,v_3)\\
&\xRightarrow{\lambda} (v_2,p_3,v_3)||(v_2,\varepsilon,v_3) \xRightarrow{\lambda} (v_2,\varepsilon,v_3)||(v_2,\varepsilon,v_3)\\
\mathfrak{c}'_1 :\ (v_0,p_0,v_3) &\xRightarrow{a} (v_1,p_2,v_3)||(v_1,p_2,v_3) \xRightarrow{b} (v_1,p_2,v_3)||(v_2,p_5,v_3).(v_3,p_4,v_3)\\
&\xRightarrow{\lambda} (v_1,p_2,v_3)||(v_2,\varepsilon,v_3).(v_3,p_4,v_3) \xRightarrow{b} (v_2,p_5,v_3).(v_3,p_3,v_3)||(v_2,\varepsilon,v_3).(v_3,p_4,v_3)\\
&\xRightarrow{\lambda} (v_2,\varepsilon,v_3).(v_3,p_3,v_3)||(v_2,\varepsilon,v_3).(v_3,p_4,v_3)
\end{aligned}$$

(c) Derivation(s) in the Combined Abstraction if rules $\mathcal{R}^0$ would be ommitted



(d) A Protocol Violation

$$\begin{aligned}
&(v_0,p_0,v_3) \xRightarrow{a} (v_1,p_2,v_3)||(v_1,p_2,v_3) \xRightarrow{b} (v_1,p_2,v_3)||(v_2,p_5,v_3).(v_3,p_4,v_3) \xRightarrow{\lambda} \underline{(v_2,p_2,v_3)||(v_2,p_5,v_3).(v_3,p_4,v_3)}\\
&\xRightarrow{b} (v_3,p_5,v_3).(v_3,p_3,v_3)||(v_2,p_5,v_3).(v_3,p_4,v_3) \xRightarrow{\lambda} \underline{(v_3,p_5,v_3).(v_3,p_3,v_3)||(v_3,p_5,v_3).(v_3,p_4,v_3)}\\
&\xRightarrow{\lambda} (v_3,p_5,v_3).(v_3,p_3,v_3)||(v_3,\varepsilon,v_3).(v_3,p_4,v_3) \xRightarrow{\lambda} (v_3,p_5,v_3).(v_3,p_3,v_3)||(v_3,p_4,v_3)\\
&\xRightarrow{\lambda} (v_3,p_5,v_3).(v_3,p_3,v_3)||(v_3,\varepsilon,v_3) \xRightarrow{\lambda} (v_3,p_5,v_3).(v_3,p_3,v_3) \xRightarrow{\lambda} (v_3,\varepsilon,v_3).(v_3,p_3,v_3) \xRightarrow{\lambda} (v_3,p_3,v_3)\\
&\xRightarrow{\lambda} (v_3,\varepsilon,v_3) \xRightarrow{\lambda} \varepsilon
\end{aligned}$$

(e) A Counterexample derived in the Combined Abstraction

Fig. 9. On the Necessity of the Rules $\mathcal{R}^0$

derivation without using $\mathcal{R}^0$-rules ends with one of the process-algebraic expressions $(v_2,\varepsilon,v_3)||(v_2,\varepsilon,v_3)$, $(v_2,\varepsilon,v_3)||(v_2,\varepsilon,v_j).(v_j,p_4,v_3)$, $(v_2,\varepsilon,v_i).(v_i,p_3,v_3)||(v_2,\varepsilon,v_3)$, or $(v_2,\varepsilon,v_i).(v_i,p_3,v_3)||(v_2,\varepsilon,v_j).(v_j,p_4,v_3)$, $i,j = 0,1,3$. Fig. 9(d) shows a derivation in the abstract system behavior as sequences of cactus stacks and the corresponding protocol state changes. This execution demonstrates a counterexample. Thus, without the $\mathcal{R}^0$-rules, false positives would be obtained. The reason is that the process-algebraic term $(v_1,p_1,v_3)\,||\,(v_2,p_5,v_2).(v_2,p_5,v_3)$ (after two steps in $\mathfrak{c}'_0$) does not reflect the boxed situation protocol state (after two steps in the execution of the abstract system behavior), since the underlined process-algebraic expression corresponds to a cactus stack where the two top stack elements would be in different protocol states although the protocol is in state v_2. The rules in $\mathcal{R}^0$ are used to establish a situation where each top stack element has the same protocol state, cf. Fig. 9(e). The first underlined derivation step used an $\mathcal{R}^0$ rule to establish the boxed situation in Fig. 9(d), the second underlined derivation step is used to establish the state after the boxed situation. Otherwise, the steps correspond exactly to those in Fig. 9(e).

For the overall approach of protocol conformance checking, false negatives can

$$
\begin{aligned}
\mathfrak{c}_0 : (v_0,p_0,v_3) &\overset{a}{\Rightarrow} (v_1,p_2,v_3)|||(v_1,p_2,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)||\underline{(v_2,p_2,v_3)} \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)||\underline{(v_3,p_2,v_3)}\\
&\overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)|||(v_3,\varepsilon,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3) \overset{b}{\Rightarrow} \underline{(v_2,p_5,v_3).(v_3,p_3,v_3)} \overset{\lambda}{\Rightarrow} \underline{(v_3,p_5,v_3).(v_3,p_3,v_3)}\\
&\overset{\lambda}{\Rightarrow} (v_3,\varepsilon,v_3).(v_3,p_3,v_3) \overset{\lambda}{\Rightarrow} (v_3,p_3,v_3) \overset{\lambda}{\Rightarrow} (v_3,\varepsilon,v_3) \overset{\lambda}{\Rightarrow} \varepsilon\\
\mathfrak{c}_0 : (v_0,p_0,v_3) &\overset{a}{\Rightarrow} (v_1,p_2,v_3)|||(v_1,p_2,v_3) \overset{b}{\Rightarrow} (v_1,p_2,v_3)||\underline{(v_2,p_5,v_2).(v_2,p_2,v_3)}\\
&\overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)|||(v_2,\varepsilon,v_2).(v_2,p_2,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)|||(v_2,p_2,v_3)\\
&\overset{b}{\Rightarrow} (v_1,p_2,v_3)|||(v_3,p_5,v_3).(v_3,p_4,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)||(v_3,\varepsilon,v_3).(v_3,p_4,v_3)\\
&\overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)|||(v_3,p_4,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3)|||(v_3,\varepsilon,v_3) \overset{\lambda}{\Rightarrow} (v_1,p_2,v_3) \overset{b}{\Rightarrow} \varepsilon \text{ (as in } \mathfrak{c}_0)
\end{aligned}
$$

Fig. 10. Spurious False Negatives of the Example in Fig. 9

be classified into the following categories:

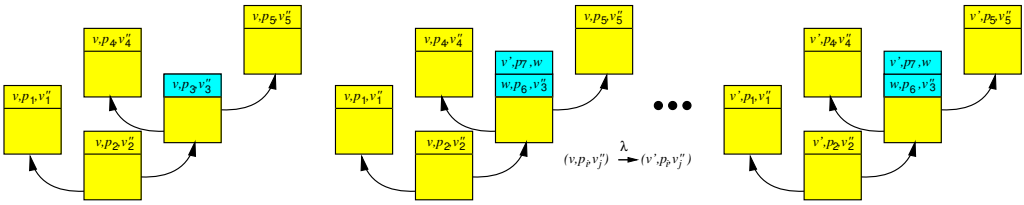
- *Real false negatives*: Because the source code abstractions are created without any data-flow or detailed control-flow analysis, it is possible that a trace could be contained in the component abstraction that does not correspond to any execution path of the implementation.
- *Spurious false negatives*: We construct an approximated intersection of the languages described by the system and by the considered inverted protocol. Therefore, it is possible to get false negatives.

The rules $\mathcal{R}^0$ may lead to spurious false negatives, cf. Fig. 10. The counterexample $\mathfrak{c}_0$ produces the interactions *ab* and has therefore too few interactions. In contrast, counterexample $\mathfrak{c}_1$ produces too many interactions *abbb*.

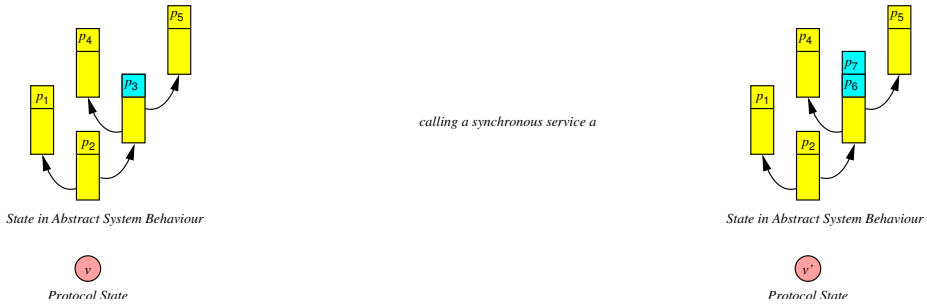
The counterexamples can be discovered as spurious false negatives by running the interactions on both the abstract system behaviour and on the protocol. Thus, this would be analogous to a CEGAR-loop. However, both counterexamples stems from a misuse of the rules in $\mathcal{R}^0$: these rules are introduced only to obtain a consistent view on the protocol states as discussed in Fig. 9. However, the spurious false negatives used these rules on other places (see the underlined expressions). Counterexample $\mathfrak{c}_0$ used the rules in $\mathcal{R}^0$ just for changing the protocol state of the second parallel thread which enabled its termination in the Combined Abstraction. Therefore, there is one interaction b less than in the real counterexample. Counterexample $\mathfrak{c}_1$ is different because here the second thread in the Combined Abstraction proceeds and changes thereby its protocol state, and the $\mathcal{R}^0$-rules are *not* used to change the protocol state in the first thread. The second thread then produces two interactions b while the protocol state in the first thread is still v_1. With one interaction of b this can be derived to ε (which again uses a transition rule of $\mathcal{R}^0$).

4 Reducing the Number of Spurious False Negatives

The example in Fig. 9 and 10 shows that spurious false negatives may stem from inappropriate application of $\mathcal{R}^0$-rules. The main idea is that when in a derivation step in the Combined Abstraction a protocol state changes, then the $\mathcal{R}^0$-rules are used to change the protocol state of the other top stack elements in the cactus stack in order to obtain a consistent view as described in the discussion of Fig. 9. Furthermore, this is the only place where $\mathcal{R}^0$ rules are being applied. Fig. 11(a) demonstrates the idea: The first step (a call of a synchronous service a) changes a protocol state into v'. Thus, there is one stack with a top element that has protocol



(a) Cactus Stack Representation of the Combined Abstraction



(b) Corresponding (Simultaneous) Execution in the Abstract System Behavior and the Protocol

Fig. 11. Round-Robin Search Strategy

state v' while the others still have protocol state v. Before applying a rule different from $\mathcal{R}^0$-rules, the $\mathcal{R}^0$-rules are used to convert the protocol states on the other top elements of the cactus stack also into v'. Fig. 11(b) shows the corresponding situation in the system abstraction and the protocol states before and after the call of a. For an asynchronous call b the approach works analogously. We call this search strategy *Round-robin*.

Now, we show that the Round-robin search strategy never leads to false positives, i. e., if there derivation $p_0 \overset{w}{\Rightarrow} \varepsilon$ in the abstract system behaviour such that $v_0 \overset{w}{\Rightarrow} v_f$ in the protocol for a final state v_f, then it is possible to construct a derivation $(v_0, p_0, v_f) \overset{w}{\Rightarrow} \varepsilon$ in the Combined Abstraction using the Round-robin search strategy.

The proof requires some technical definitions and lemmas. Their meaning will be explained using the cactus stack view as in Fig. 11. The *multiset of heads* of a process-algebraic expression t denotes the multiset of elements on the top of the stacks in the cactus stack representation of t (a multiset is required because there might be two stacks with the same top elements). Formally, a cactus stack corresponds to a process-algebraic expression of the form $(p_1.u_1||\ldots||p_n.u_n).u_{n+1}$, where $n > 0$, $u_1, \ldots, u_{n+1} \in PEX(Q)$. Note that process-rewrite rules are only applied to one of the atomic processes $p_1, \ldots, p_n \in Q$. Then, the set of heads can be inductively defined by:

$$H(p) \hat{=} \{\{p\}\}$$
$$H(t_1.t_2) \hat{=} H(t_1)$$
$$H(t_1||t_2) \hat{=} H(t_1)||H(t_2)$$

As one transition is a synchronization, one must define which threads could be explicitly synchronized. It is only possible to sychronize two threads if the top element of a stack in the cactus stack forks into a stack with one element. We call

pairs of such top stack elements *synchronization possibilities*.

Property 1 states that any rule is either applied to a top frame of a cactus stack or it synchronizes two threads (i. e., it merges two stacks into one).

Property 1 *If in a PRS* Π *a rule* $\delta : t_1 \xrightarrow{b} t_2$ *is applied to process-algebraic expression* t *(i. e.,* $t \overset{b}{\Rightarrow} t'$*), then there is a head* $h \in H(t)$ *or a synchronization possibility* $h \in S(t)$*, s. t.* t' *is obtained from* t *by replacing* h *by* h' *by the same rule* δ*, where* $h \overset{b}{\Rightarrow} h'$*.*

A *protocol state of a process-algebraic term* is the set of protocol states. The protocol state of a triple (v, p, v') is v. The set $\mathrm{ps}(t) = \{v : (v, p, v') \in H(t)\}$ denotes the set of protocol states of the top elements of a cactus stack.

The following theorem states the property shown in Fig. 11. Whenever there is a situation as in Fig. 11(b) that terminates (i. e., a cactus stack of the abstract system behavior becomes empty) with interactions w and the current protocol state v can be transformed with w into a final state, then there is a cactus stack s in the Combined Abstraction (as shown in Fig. 11) where v is the protocol state in the top elements of all stacks such that s terminates with the interactions w.

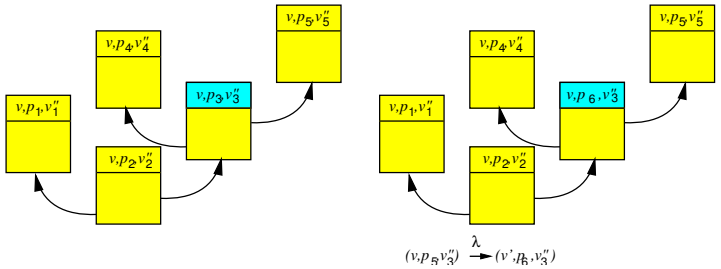
Theorem 4.1 *Let* $\Pi_{S,C}$ *be a abstract system behavior (considering component* C*),* P_C *be the inverted protocol of component* C*, and* $w \in \Sigma^*$*, such that* $t \overset{w}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ *for a term* $t \in PEX(Q_{\Pi_{S,C}})$ *and* $v \overset{w}{\Rightarrow}_{P_C} f$ *for a protocol state* $v \in Q_{P_C}$ *and a final protocol state* $f \in F_{P_C}$*. Then there is a process-algebraic expression* $s \in Q_{\Pi_S^C}$ *over the Combined Abstraction such that* $s \overset{w}{\Rightarrow} \varepsilon$ *and* $|ps(s)| = 1$*, i. e., the protocol states in the heads are equal.*

The proof of Theorem 4.1 requires some technical definitions and lemmas. Let $s \in PEX(Q_{\Pi_S^C})$ be a process-algebraic expression over the Combined Abstraction Π_S^C. As it can be seen from Fig. 11, the cactus stacks of the Combined Abstraction can be transformed into the cactus stacks of the abstract system behaviour by projection on the states of the abstract system behaviour. Thus, the process-algebraic expression $\mathrm{F}(s) \in PEX(Q_{\Pi_{S,C}})$ over the abstract system behavior $\Pi_{S,C}$ is defined as:

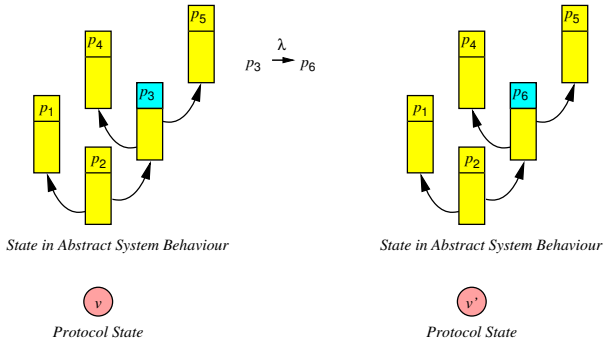
$$\begin{aligned} \mathrm{F}((v,p,v')) &= p && \forall (v,p,v') \in Q_{\Pi_S^C} \\ \mathrm{F}(s.s') &= \mathrm{F}(s).\mathrm{F}(s') && \forall s, s' \in PEX(Q_{\Pi_S^C}) \\ \mathrm{F}(s||s') &= \mathrm{F}(s)||\mathrm{F}(s') && \forall s, s' \in PEX(Q_{\Pi_S^C}) \end{aligned}$$

E. g., $\mathrm{F}((v_1,p_2,v_5)||(v_1,p_3,v_2).(v_2,p_4,v_5)) = p_2||p_3.p_4$. Furthermore, let $\mathrm{F}^{-1}(t) = \{s|\mathrm{F}(s) = t\}$.

Fig. 12 visualizes the statement in Lemma 4.2. If in the abstract system behavior no interaction happens in the next step (i. e., a λ-rule is applied) and the top elements of the stacks in the corresponding cactus stack of the Combined Abstraction all contain the same protocol state v, then the Combined Abstraction can perform an analogous step. The next cactus stack in the derivation of the Combined Abstraction corresponds to the next cactus stack in the execution of the abstract system behavior. Further the protocol state of the top elements of the stacks in the



(a) Cactus Stack Representation of the Combined Abstraction



(b) Corresponding (Simultaneous) Execution in the Abstract System Behavior and the Protocol

Fig. 12. Transitions in the Combined Abstraction and the Abstract System Behavior without Interactions

next cactus stack in the Combined Abstraction is still v. Lemma 4.2 is an inductive variant of the situation in Fig. 12.

Lemma 4.2 *Let $t, t' \in PEX(Q_{\Pi_{S,C}})$ such that $t \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$. Then, for all $s' \in F^{-1}(t')$, there is an $s \in F^{-1}(t)$ such that the following properties are satisfied.*

(i) *$s \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'$ using only rules $\delta \in \mathcal{R}_C^\lambda \cup \mathcal{R}_{Seq}^\lambda \cup \mathcal{R}_{PFork}^\lambda \cup \mathcal{R}_{PSync}^\lambda$.*

(ii) *$(v,p,v') \in H(s)$ iff there is a $p' \in Q_{\Pi_{S,C}}$ such that either $(v,p',v') \in H(s')$ or $(v,p,v) \in H(s')$.*

Remark: (ii) implies that $ps(s) = ps(s')$, i. e., no state change in the protocol happens.

Proof. Straightforward induction on the construction of $t \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$. □

Lemma 4.3 states that the Round-robin derivation can always be constructed, if there is exactly one protocol state change. It states that if the abstract system behaviour can execute a derivation step as depicted in Fig. 11(b) with a protocol state change, then there is derivation according to the Round-robin strategy as depicted in Fig. 11(a). Obviously, a similar situation could be visualized for calling asynchronous services and synchroniziations.

Lemma 4.3 *Let $t_1, t_2 \in PEX(Q_{\Pi_{S,C}})$ be two process-algebraic expressions over the abstract system behavior $\Pi_{S,C}$ such that $t_1 \stackrel{a}{\Rightarrow}_{\Pi_{S,C}} t_2$ for an $a \in \Sigma$ by application of a single rule δ. If the protocol P_C contains a transition rule $v \stackrel{a}{\rightarrow}_{P_C} v'$, then for any*

$s_2 \in F^{-1}(t_2)$ satisfying $ps(s_2) = \{v'\}$, there is a $s_1 \in F^{-1}(t_1)$ such that the following properties are satisfied:

(i) $ps(s_1) = \{v\}$

(ii) $s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s' \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_2$ *where* $s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s'$ *uses a single rule* $\delta \in \mathcal{R}_C^1 \cup \mathcal{R}_{Seq}^1 \cup \mathcal{R}_{PFork}^1 \cup \mathcal{R}_{PSync}^1$ *and* $s' \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_2$ *only uses rules δ which are corresponding sleep rules of* $v \stackrel{a}{\rightarrow}_{P_C} v'$.

Proof. (of Lemma 4.3)
Case 1: $\delta \hat{=} (p \stackrel{a}{\rightarrow}_{\Pi_{S,C}} p')$ or $\delta \hat{=} (p \stackrel{a}{\rightarrow}_{\Pi_{S,C}} p' || p'')$ or $\delta \hat{=} (p || p'' \stackrel{a}{\rightarrow}_{\Pi_{S,C}} p')$. Then, there must be a protocol state $v'' \in Q_{P_C}$ such that $(v', p', v'') \in H(s_2)$. Suppose that $H(s_2) \setminus \{(v', p', v'')\} = \{(v', p_1, v_1), \ldots, (v', p_n, v_n)\}$. Then a derivation $s'_n \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_{n-1} \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} \ldots \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_1 \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s'_0 = s_2$ where s'_{i-1} is the result of the application of the rule $(v, p_i, v_i) \stackrel{\lambda}{\rightarrow}_{\Pi_S^C} (v', p_i, v_i)$ at head (v, p_i, v_i) of term s'_i, $i = n, \ldots, 1$. Thus, only corresponding sleep rules of $v \stackrel{a}{\rightarrow}_{\Pi_S^C} v'$ are applied, and $(v', p', v'') \in H(s'_n)$ is the only atomic process with a protocol state different from v. By induction on the construction of $t_1 \stackrel{a}{\Rightarrow}_{\Pi_{S,C}} t_2$, one can prove that $s \stackrel{a}{\Rightarrow}_{\Pi_S^C} s'_n$ using $(v, p, v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v', p, v'')$, $(v, p, v'')||(v, p', v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v', p'', v'')$ for a $v'' \in Q_{P_C}$, or $(v, p, v'') \stackrel{a}{\rightarrow}_{\Pi_S^C} (v', p', v'')|||(v', r'', v'')$, respectively.
Case 2: $\delta = p \stackrel{a}{\rightarrow}_{\Pi_{S,C}} p'||p''$. Then there is a protocol state $v'' \in Q_{P_C}$ such that $(v', p', v'') \in H(s_2)$, and $(v', p'', v'') \in H(s_2)$. The rest of the proof is analogous except that $H(s_2) \setminus \{\{(v', p', v''), (v', p'', v'')\}\}$ is used. □

The proof of Theorem 4.1 is basically an inductive application of Lemmas 4.2 and 4.3.

Proof. (of Theorem 4.1)
Induction on w:
$w = \lambda$: By Lemma 4.2, there is a $s \in \mathrm{F}^{-1}(t)$ such that $s \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ and, since $v \stackrel{\lambda}{\Rightarrow}_{P_C} f$ implies $v = f$, $H(s) = \{\{f\}\}$.
$w = ax$: for an $a \in \Sigma$ and $x \in \Sigma^*$.
Then, $t \stackrel{ax}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ has the form $t \stackrel{\lambda}{\Rightarrow}_{\Pi_{S,C}} t_1 \stackrel{a}{\Rightarrow}_{\Pi_{S,C}} t_2 \stackrel{x}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$, and $v \stackrel{w}{\Rightarrow}_{P_C} f$ has the form $v \stackrel{a}{\Rightarrow}_{P_C} v' \stackrel{x}{\Rightarrow}_{P_C} f$.
By induction hypothesis, there is an $s_2 \in PEX(Q_{\Pi_S^C})$ such that $s_2 \stackrel{a}{\Rightarrow}_{\Pi_S^C} \varepsilon$ and $|ps(s_2)| = 1$. By Lemma 4.3, there is an $s_1 \in PEX(Q_{\Pi_S^C})$ such that $s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s_2$, where $\Rightarrow_{\Pi_S^C}$ is constructed according to Round-robin reachability. By Lemma 4.2, there is an $s \in PEX(Q_{\Pi_S^C})$ such that $s \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_1$ only using rules of $\mathcal{R}^0$ and $|\mathrm{ps}(s)| = 1$. Thus, $s \stackrel{\lambda}{\Rightarrow}_{\Pi_S^C} s_1 \stackrel{a}{\Rightarrow}_{\Pi_S^C} s_2 \stackrel{x}{\Rightarrow}_{\Pi_S^C} \varepsilon$. □

The following corollary states that any non-spurious counterexample can be found using the Round-robin strategy.

Corollary 4.4 *If $w \in L(\Pi_{S,C}) \cap L(P_C)$, then there is a Round-robin reachability*

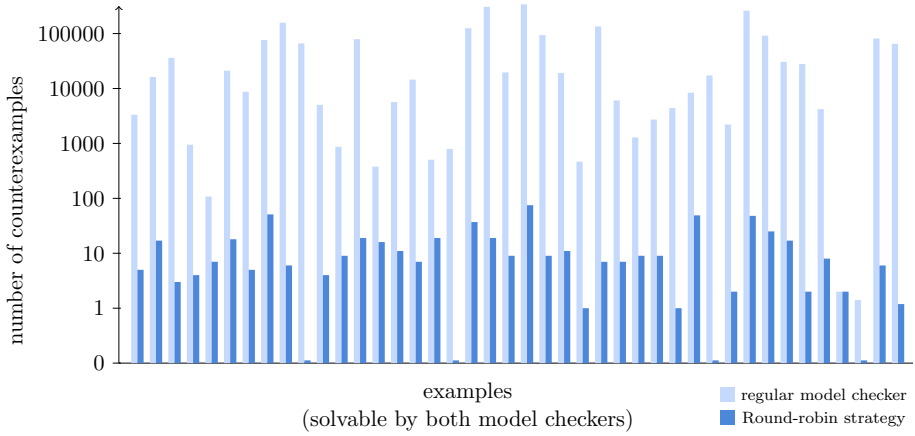


Fig. 13. Results of Case Study

derivation $t \overset{w}{\Rightarrow}_{\Pi_S^C} \varepsilon$ *in the Combined Abstraction* Π_S^C*, where* $t \in Q_{\Pi_S^C}$ *is the initial state of* Π_S^C*.*

Hence, the reachability problem can be solved by using the Round-robin reachability and will create fewer false negatives. This leads to a better applicability because a component developer or quality management representative has to check a lower number of counterexamples. Moreover, because we cut branches during the verification, it will probably be finished faster.

5 Case Study

We have conducted two industrial case studies. Both rarely use parallelism. Therefore, we generate randomly examples containing a high rate of parallel transition rules ($\approx 20\%$) for checking the benefits of the Round-robin reachability. Regarding our earlier experiences to the industrial case this is a high rate. This results in more complex models, that are harder to solve. Here, we model check the complete models, to calculate all contained counterexamples (timeout 15 min). We use a single threaded prolog implementation (SWI-Prolog version 5.6.47 [40]) on a 32-bit Linux machine (2200 MHz).

Two experiences are made while evaluating the case study.

- If many parallel transition rules are contained in the model and derivable in each other, then the number of counterexamples generated by the model checker without Round-robin reachability explodes. This is caused by the many interleaving possible while ignoring the protocol states encoded within the Combined Abstraction. These counterexamples are reduced dramatically. The average reduction rate for the examples, that are solvable by both model checkers, is 97.16%. The results are shown in Figure 13.
- The Round-robin reachability is significantly faster. This was observed as 58.93% of the runs without Round-robin reachability calculate less counterexamples than

the Round-robin reachability within the time constraints. This is caused by the behavior of the Round-robin reachability enabling the evaluation of a derivation path only if precisely one action rule was applied. Thereafter, all protocol states are synchronized. In contrast, the trivial evaluation approach evaluates each path.

6 Related Work

Many works on static protocol-checking of components consider local protocol checking on FSMs. The same approach can also be applied to check protocols of objects in object-oriented systems. The idea of static type checking by using FSMs goes back to Nierstrasz [30]. His approach uses regular languages to model the dynamic behavior of objects, which has the same power as finite state approaches. Therefore, the approach cannot handle recursive call-backs. In [27] object-life cycles for the dynamic exchange of implementations of classes and methods using a combination of the bridge/strategy pattern are considered. This approach also bases on FSMs. The approach comprises dynamic as well as static conformance checking. Tenzer and Stevens [38] investigate approaches for checking object-life cycles. They assume that object-life cycles of UML-classes are described using UML state diagrams and that for each method of a client, there is a FSM that describes the calling sequence from that method. In order to deal with recursion, Tenzer and Stevens add a rather complicated recursion mechanism to FSMs. It is not clear whether this recursion mechanism is as powerful as pushdown automata. All these works consider pure sequential systems.

[34] provides a good overview on the state of the art in component models. The case study *CoCoME* [22] is used to compare component models. This case study only requires non-recursive procedures and bounded concurreny. Therefore, all finite state approaches for protocol conformance checking are suitable for this case study.

Most of the component models discussed in [34] include behavioral protocols and check their compatibility. Furthermore, most models follow a top-down approach. *KobrA* [4] uses activity diagrams to specify abstract component behaviors. Although not fully formal, it is possible to specify bounded concurrent execution, but unbounded recursion cannot be modeled using activity diagrams. The *KobrA* approach seems not to include automatic conformance checking. The *Java/A* component model [24] is similar since they also use activitiy diagrams for specifying abstract component behaviors. *Rich Services* [18] use message sequence charts to specify component behaviors and their interactions. It is possible to model bounded concurrency and bounded recursion. This approach is semi-formal and seems not being mechanized. In contrast to these two component models, *rCOS* [16] separates component protocols from component behaviors. Protocols are specified by guarded state diagrams which is an extension of finite state machines. They derive CSP-expressions to specify the abstract system behavior. Model checking is used for automatic protocol conformance checking as well as for checking liveness conditions. By using CSP the approach is able to deal with concurrency. [10] models the

CoCoME case study with *Focus* and *AutoFocus*. The interactions between components in a component-based system are specified using message sequence charts. The component behavior is described by finite state machines that have three kinds of transitions: outgoing messages, incoming messages, and internal transitions. The interactions and component behavior are glued to construct the abstract system behavior. Since this is finite state, standard model checkers can be used to check properties including protocols. The *Grid Component Model* [15] uses similar finite state machines as [10]. However, these can be parameterized by the number of parallel threads.

The *DisCComp* model [3] is similar powerful as the approach described in this work. They also distinguish synchronous and asynchronous messages with a similar semantics as ours. The *DisCComp* model uses OCL to specify contracts for components and a Turing-complete model to specify the component behavior. For tool support, the approach uses similar techniques as [18,10]. The *CoIn*-model [44] specifies component behavior by finite state machines. The interactions with other components are made explicit on the transitions. These automata are composed to finite state machines specifying the interactions in a component-based system. They use the abstract system behavior for model-checking conditions specified as LTL-formulas. Therefore, this approach can be used for protocol conformance checking as protocols can be considered as a special case of model-checking. By using a finite state approach, *CoIn* is restricted to component based systems with bounded recursion and bounded concurrency. The strength of *CoIn* is the ability to automatically build the abstract component behavior of composed component C from the component behaviors of the components of C.

The *SOFA component model* [12] and the *Fractal* component model [11] use process-algebraic expressions to specifiy component behavior. Compatibility of component interactions is checked automatically. The process-algebraic expressions are regular expressions (extended by a parallel-operator) over incoming and outgoing messages. It seems that their expressiveness is equivalent to the kind of finite state machines in [10]. The Palladio component model [25] uses finite state machines extended by effects to specifiy component behaviors. The effects specify updates on quantitative properties that can be used for predicitions (e. g., performance).

Although [34] gives a good overview of the state of the art, there are some other works on protocol conformance checking to be mentioned. [41] uses abstract component behaviors specified as finite state machines to generate adaptors that ensure protocol conformance. Schmidt et al. [21] propose an approach for protocol checking of concurrent component-based systems but recursion is not considered. In [1,5,2] process algebras such as CSP are used to describe component behaviors. These approaches are more powerful than finite state and pushdown system based approaches. However, mechanized checking requires some restrictions on the specification language. For example, [1] uses a subset of CSP that allows only the specification of finite processes. At the end the conformance checking is reduced to checking finite state machines similar to [21]. Many works use process algebras as abstractions for the formal (behavioral) analysis of e. g., BPEL applications. [19]

uses CSP for modeling the abstract behavior of components while [37] uses CCS. These process algebras are similar to PAN considered in our work. Mechanization of protocol conformance checking was not an aim of these two works.

Context-free grammars for modeling abstract component behaviors and abstract system behaviors are proposed in [42,43]. However, there is no operation such as a shuffle on context-free languages. Therefore, the approach can only deal with sequential systems. The model-checking problem is discussed in [9]. [7,6] unify the power of pushdown automata with unbounded concurrency by using process rewrite systems. The algorithm in [28] is used for the approximate model-checking. There are further works on model checking PRS: While [32] generates overapproximations of the execution paths, underapproximations of the reachable configurations are computed in [26] (bounded model checking).

To the best of our knowledge no other work on approximate model checking of PRS exists that use an a priori approach to reduce the number of false negatives.

7 Conclusions and Future Work

Our verification approach ensures properties based on interaction protocols of programs which include unbounded recursion and unbounded parallelism. This leads to a higher quality of software, because the protocol conformance can be checked before the deployment. Every error will be found.

This work has shown how the special properties of the representation (defined in earlier works) can be used to reduce the number of false negatives. We call this approach Round-robin reachability, because it balances the derivation steps applied on each parallel term. This improvement reduces the costs of the quality check, because fewer counterexamples have to be reviewed to find the real errors. We assume that, most spurious counterexamples might be removed in an industrial environment.

Although this improved verification process can not reduce the complexity of the verification in a general case, we assume that in an industrial setting the verification will be finished much faster. To check this assumption is part of our future work.

In a more general context, it is shown here, that an overapproximated model checking of LTL formula seems to be possible, while using the transformation of a LTL formula into a Büchi automata. Thereafter, the automata might be intersected with the PRS. Proving this assumption is part of future work.

Currently, we validate our approach in an industrial case study of component-based systems written in Python and C/C++. We can create abstractions of source codes written in Python, already. Presently, we are creating a generator of abstractions for C/C++ source code. Early results show that our approach is capable of finding errors or unexpected behavior in real programs. In future work we will create abstractions of BPEL and PHP source codes, considering Java is also planned.

We thank the anonymous referees for their helpful comments.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[2] P. Andre, G. Ardourel, and C. Attiogbe. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In *Software Composition: 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, page 2. Springer, 2007.

[3] A. Appel, S. Herold, H. Klus, and A. Rausch. *Modelling the CoCoME with DisCComp*, chapter 11 in [34], pages 267–296. Springer, 2009.

[4] C. Atkinson, P. Bostan, D. Brenner, G. Falcone, M. Gutheil, O. Hummel, M. Juhasz, and D. Stoll. *Modeling Components and Component-Based Systems in KobrA*, chapter 4 in [34], pages 54–84. Springer, 2008.

[5] C. Attiogbe, P. Andre, and G. Ardourel. Checking component composability. *LNCS*, 4089:18, 2006.

[6] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel BPEL systems. *IEEE Sixth European Conference on Web Services (ECOWS '08)*, 0:81–91, 2008.

[7] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel component-based systems. In Michel R. V. Chaudron, Clemens A. Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, volume 5282 of *LNCS*, pages 163–179. Springer, October 2008.

[8] Andreas Both and Wolf Zimmermann. On more predictable implementations of reliable workflows in service-oriented architectures. *IEEE Seventh European Conference on Web Services (ECOWS '09)*, November 2009.

[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR'97: Proc. of the 8th Int. Conf. on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[10] M. Broy, J. Fox, F. Hlzl, D. Koss, M. Kuhrmann, M. Meisinger, B. Penzenstadler, S. Rittmann, B. Schtz, M. Spichkova, and D. Wild. *Service-Oriented Modeling of CoCoME with Focus and AutoFocus*, chapter 8 in [34], pages 178–206. Springer, 2008.

[11] A. Bulej, T. Bures, T. Coupaye, M. Decky, P. Jezek, P. Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. *CoCoMe in Fractal*, chapter 14 in [34], pages 357–387. Springer, 2009.

[12] T. Bures, M. Decky, P. Hnetynka, J. Kofron, P. Parizek, F. Plasil, T. Poch, O. Sery, and P. Tuma. *CoCoME in SOFA*, chapter 15 in [34], pages 388–417. Springer, 2008.

[13] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92: Proc. of the 3rd Int. Conf. on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1992.

[14] O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *CONCUR'94: Proc. of the 5th Int. Conf. on Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 1994.

[15] A. Cansado, D. Caromel, L. Henrio, E. Madelaine, M. Rivera, and E. Salageanu. *A Specification Language for Distributed Components Implemented in GCM/ProActive*, chapter 16 in [34], pages 418–448. Springer, 2008.

[16] Z. Chen, A. H. Hannousse, D. V. Hung, I. Knoll, X. Li, Z. Liu, Y. liu, Q. Nan, J. C. Okika, A. P. Ravn, V. Stolz, L. Yang, and N. Zhan. *Modelling with Relational Calculus of Object and Component Systems – rCOS*, chapter 6 in [34], pages 116–145. Springer, 2008.

[17] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[18] B. Demchak, V. Ermagan, E. Farcas, T. Huang, I. Krger, and M. Menarini. *A Rich Services Approach to CoCoME*, chapter 5 in [34], pages 85–115. Springer, 2008.

[19] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Model-based analysis of obligations in web service choreography. In *AICT/ICIW*, page 149. IEEE Computer Society, 2006.

[20] J. Freudig, W. Löwe, R. Neumann, and M. Trapp. Subtyping of context-free classes. In *Proc. 3rd White Object Oriented Nights*, 1998.

[21] H. W. Schmidt, B. J. Krämer, I. Poernemo, and R. Reussner. Predictable component architectures using dependent finite state machines. In *Proc. of the NATO Workshop Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2002.

[22] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. *CoCoMe – The Common Component Modeling Example*, chapter 3 in [34], pages 16–53. Springer, 2008.

[23] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[24] A. Knapp, S. Janisch, R. Hennicker, A. Clark, S. Gilmore, F. Hacklinger, H. Baumeister, and M. Wirsing. *Modelling the CoCoME with the Java/A Component Model*, chapter 9 in [34], pages 207–238. Springer, 2008.

[25] R. Krogmann and R. Reussner. *Palladio – Predicition of Performance Properties*, chapter 12 in [34], pages 297–326. Springer, 2008.

[26] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. Technical Report 1598, Computer Sciences Department, University of Wisconsin, 2007.

[27] W. Löwe, R. Neumann, M. Trapp, and W. Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS 29 – Technology of Object-Oriented Languages and Systems*, pages 351–360. IEEE, 1999.

[28] Richard Mayr. Combining petri nets and pa-processes. In *TACS'97: Proc. of the Third Int. Symposium on Theoretical Aspects of Computer Software*, pages 547–561, London, UK, 1997. Springer.

[29] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1-2):264–286, 2000.

[30] Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[31] Pavel Parizek and Frantisek Plasil. Modeling of component environment in presence of callbacks and autonomous activities. In Richard F. Paige and Bertrand Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 2–21. Springer, 2008.

[32] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 254–257. Springer, 2007.

[33] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[34] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example*. Number 5133 in Lecture Notes in Computer Science. Springer, 2008.

[35] Wolfgang Reisig. Modeling- and Analysis Techniques for Web Services and Business Processes. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems: 7th IFIP WG 6.1 Int. Conf., FMOODS 2005, Athens, Greece, June 15-17, 2005. Proc.*, volume 3535 of *LNCS*, pages 243–258. Springer, May 2005.

[36] Ralf H. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In Bill Grosky, Frantisek Plasil, and Ales Krenek, editors, *Proc. of the 29th Annual Conf. in Current Trends in Theory and Practice of Informatics (SOFSEM 2002), Milovy, Czech Republic*, volume 2540 of *LNCS*, pages 20–40. Springer-Verlag, Berlin, Germany, November 2002.

[37] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. *International Conference on Web Services*, 0:43, 2004.

[38] J. Tenzer and P. Stevens. Modelling recursive calls with uml state diagrams. In *6th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2003.

[39] T. Touili. Constrained reachability of process rewrite systems. In *Theoretical Aspects of Computing - ICTAC 2009*, number 5684 in Lecture Notes in Computer Science, pages 307–321. Springer, 2009.

[40] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[41] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[42] W. Zimmermann and M. Schaarschmidt. Model checking of client-component conformance. In *2nd Nordic Conf. on Web-Services*, number 008 in Mathematical Modelling in Physics, Engineering and Cognitive Sciences, pages 63–74, 2003.

[43] Wolf Zimmermann and Michael Schaarschmidt. Automatic checking of component protocols in component-based systems. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *LNCS*, pages 1–17. Springer, 2006.

[44] B. Zimmerova, P. Varekova, N. Benes, I. Cerna, L. Brim, and J. Sochor. *Component Interaction Automata Approach (CoIn)*, chapter 7 in [34], pages 146–177. Springer, 2009.

A Implemented Verification Framework

To enable the use of our verification process, we have implemented a component-based architecture. While implementing this framework several requirements have to be complied. The most important ones are: (i) The industrial partner demands that no source code will be transferred over the internet (neither decrypted nor encrypted, (ii) the different steps should be deployable in different locations leading to a better usage rate (particularly with regard to the model checker), and (iii) the parts of the verification process should be able to be improved separately.

To realize the case study and allow future use, we decided to implement a web service based architecture. This ensures an independent development of the components needed for the process. The architecture divides the tasks in four parts, which are similar to the steps of the verification process (cf. Fig. 2): It consists of components implementing the following interfaces:

- *Abstractions (A)*: generate the single component abstractions and compose them to a Π_S as requested by a user (discussed in Section A.1),
- *PRS Operations (O)*: hide the different abstraction services, compute Combined Abstractions and optimize Combined Abstractions and abstract system behaviors as requested by a user (discussed in Section A.2),
- *Model Checker (B)*: solves the model checking problem, delivers the computed counterexamples to the user interfaces (discussed in Section A.3),
- *User Interfaces (U)*: provides interfaces for end users, enabling a comfortable handling of the verification process (discussed in Section A.4).

An overview is given in Figure A.1. The main task of an end user is to get an overview of the implementation and thereafter create a model checking scenario which is checked at the end.

In the following the implemented web services are described in detail.

A.1 Abstractions (short: WSA)

The abstractions of the considered components are accessible via a web service interface – called shortly *WSA*. Implementations of the WSA interface create abstractions of a given set of source code and translate them into PRS. Thereby, optimizations specific for the considered programming language are performed. The source code of the given files is never made accessible for other services, only PRS

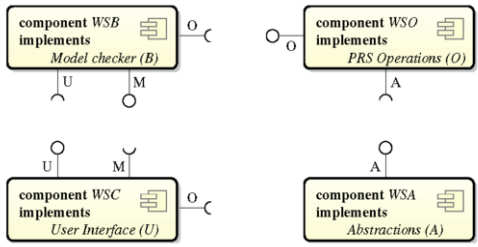


Fig. A.1. Overview of implemented architecture.

representations are provided. Thus, one main requirement – the protection of the encoded business logic – is achieved.

A.2 PRS Operations (short: WSO)

The source code analyses implemented in the WSA web services make the PRS representations of the considered source code available. They implement different optimizations to create smaller representations. It was the explicit aim to implement optimizations of the PRS. This is done by the implementation of a web service called *PRS Operations* – abbreviated *WSO*. The main purpose of this service is to create a Combined Abstraction by request and provide several optimizations for PRS. It requires a WSA implementation and implements the same interfaces as WSA. The idea behind this is a possible pipes and filters architecture. It enables to join onto optimizations implemented in future work or other groups.

Summarized, the service encapsulates the implementation of WSA. It provides the pieces of information to the user interfaces and the model checker.

Remark: It is clear that it would not be possible to compose many implementations of optimizations (WSO) in a chain, as the performance might be reduced too much. Therefore, the current implementation implements a strategy design pattern enabling the extension by new optimizations within WSO, too. ∎

A.3 Model Checker (short: WSB)

The web service *model checker* is called WSB shortly. After getting a task of the user interface, it solves the reachability problem for PAN and delivers counterexamples to the user interfaces asynchronously.

A.4 User Interfaces (short: WSC+P2)

At last a framework was implemented. It provides several alternatives for controlling and manipulating the phases of the verification process. The application is called "P2" which is an abbreviation of the German word "Protokoll-Prüfung" meaning protocol checking.

It contains a server implementation providing multi-user support and an interface for interactions with other web services. The frontend supports the following operations (cf. Figure A.2):

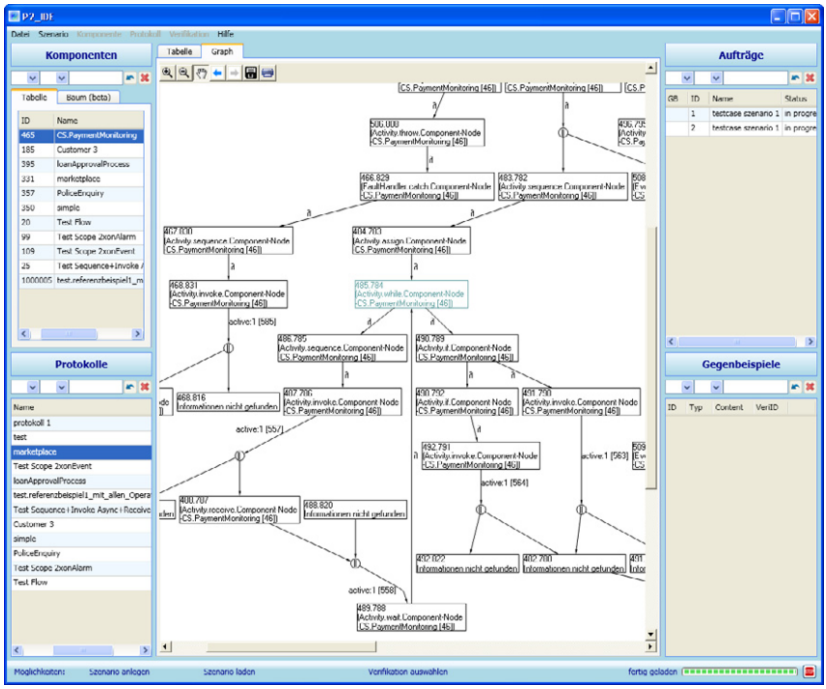


Fig. A.2. Screenshot of implemented frontend "P2".

- defining new protocols,
- showing a graphical representation of components and applications,
- defining applications and protocols (named scenarios) which should be verified,
- starting of verifications,
- evaluating delivered verifications (counterexamples) using the abstractions
- representing extended component information (e. g., author, last change, ... [6])

Moreover, an API was implemented providing Python support. It allows to start verifications and to evaluate counterexamples. This opens the option to an integration in the nightly checks of the source code at the industrial partners.

The evaluation of the counterexamples is used to improve the full set of counterexamples. If one user has marked a counterexample as invalid, all other users profit from this information as this counterexample is marked as *false negative*. The more users this counterexample exclude the less important it gets, until it disappears. This rating mechanism should help to improve the verification results.

This information gets weakened if the version (e. g., SVN revision number) or the abstraction of the source code file changes (if this information is provided by the web service "Abstraction").

[6] The set of information is flexible and depends on the configuration by the provider of the considered WSA implementation.

A.5 Summary

In this section we have presented the framework that was implemented. The framework has well defined interfaces. It is implemented using web services. Thus, it is easy extendable and adaptable. Moreover, the main property for evaluating industrial application is ensured, as the source code is always hidden.

The user of the verification process can use a graphical user interface (GUI) or a Python interface providing the API. The implementation of the web services for user interfaces omits the verification process and provides multi-user support. The GUI can represent the abstract behavior of components and composed components using a graphical representation. PRS transition rules are represented only in this way. To our experience a graphical representation of the considered counterexample is useful for the evaluation.

We are grateful to OR Soft GmbH for supporting us during the development and giving the permission to test the framework in an industrial context.