Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems

Andreas Both and Wolf Zimmermann

Institute of Computer Science, University of Halle, 06099 Halle/Saale, Germany
andreas.both@informatik.uni-halle.de,
wolf.zimmermann@informatik.uni-halle.de

Abstract. Today model checking of security or safety properties of component-based systems based on finite protocols has the flaw that either parallel or sequential systems can be checked. Parallel systems can be described often by well known Petri nets, but it is not possible to model recursive behaviour. On the other hand sequential systems based on pushdown automata can capture recursion and recursive callbacks [27], but they do not provide parallel behaviour in general.

In this work we show how this gap can be filled if process rewrite systems (introduced by Mayr [16]) are used to capture the behaviour of components. The protocols of the components interfaces specified as finite state machines can be combined to a system equal to a process rewrite system. By calculating the reachability of the fault state range one gets a trace (counterexample) which does not satisfy the properties specified by all protocols of the combined components, if any error exists.

1 Introduction and Motivation

Modern software development contains a big share of reusing previously developed software called components. Often these components are developed by third party companies and supplied in binary code or as Web Service. So it is not easy to have a look at the source code to collect the behaviour. Hence the supplier should deliver together with the component a protocol of the interfaces and an abstraction of the component which specifies the behaviour, to give us the ability to check certain properties, e. g. abortion freeness.

The protocol is in general specified as finite state machine. Today the abstractions are often specified in one of the following four ways:

- By using *Petri nets* it is possible to specify parallel behaviour like threads as well as synchronous and asynchronous method calls, but no recursion in general.
- By using *pushdown automata* (PDA) it is possible to specify recursion and synchronous method calls, but no threads or asynchronous calls in general. [27]

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 163–179, 2008.

- By using *finite state machines* (FSM) ([18, 23]) a kind of parallel behaviour can be described (cf. [22]), but no recursion in general. Normally FSM can be represented as PDA or Petri nets.
- By using *process algebras* such as CSP [1], these approaches are more powerful than FSM and PDA, but at the end the conformance checking reduces to checking FSM [12].

In (modern) component systems like OMG's CORBA, Sun's JavaEE or Microsoft's .NET parallel concepts as well as sequential concepts are permitted. Hence it will be nearly impossible to use exclusively Petri nets or pushdown automata in practice to model the behaviour of components.

Our idea is to use another representation to model the behaviour of components and component-based systems and to develop a tool which proves the absence of component protocol violations. This tool should advance the verification of a component-based system by automatically verifying the protocol conformance of its components. No expert of verification will be necessary.

In Section 2 definitions of components, protocols and abstractions will be described. Creating abstractions of a full program will be shown in Section 3. In Section 4 the so called Combined Abstraction will be introduced, which is a representation of the full component-based system in combination with a considered protocol. We show in Section 5 how a counterexample can be calculated, while solving a reachability problem. To converge our approach to real component-based systems we show in Section 6 how abstractions of single components can be constructed without knowledge of the other components in the component-based system, and how they are assembled to a full component-based system abstraction.

2 Terms and Definitions

2.1 Components and Component-Based Systems

We assume that a component is an implementation of each provided interface. It is possible, that a component uses provided interfaces, thus has required interfaces. We make no restrictions on the language nor location where the component is deployed. Of course the implementation could be inaccessible (e. g. Web Service). Our assumptions are summarized in Figure 1.

A component-based system is assembled by components which communicate only over required (and provided) interfaces. These interfaces consist of a set of functions/procedures. We allow synchronous and asynchronous interfaces. Called synchronous interfaces block their caller until the callee has been completed the call, while asynchronous interfaces start a new thread when they are called. Hence the caller can proceed without waiting for completion. We also assume, that a components interface does not contain the information if it is implemented synchronous or asynchronous in general.[1]

We allow call-backs, but no external dynamic instances of a component.

[1] For simplification we specify in our example the communication method.

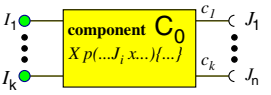

Fig. 1. Component C_0 with the required interfaces $J_1, \ldots, J_n$ and the provided interfaces $I_1, \ldots, I_n$

2.2 Protocol

A protocol of a component describes the use of all interfaces (remote use of functions) of the component. It can be used to verifiy dynamically incoming (remote) method calls, and also to verifiy the components statically. Using model checking we consider the latter in this work because it has the advantage that component-based systems are checked before they are deployed by the customer. This is a model checking problem, which is much harder to solve than verifing dynamically. Nevertheless component protocols could be used with both approaches to increase safety and security.

Creating and verifying protocols can help to ensure the restrictions of business rules. For example, using a SSO-component[2] in a system with the following actions, *a*. register and sign in, *b*. sign in, *c*. optionally change password, *d*. logout, could have the following business rule respectively protocol A formulated as regular expression: $R_A = ((a|b)c^*d)^*$.

This protocol should be obeyed by every client. We will check automatically, if a component-based system using the mentioned SSO-service protocol obeys the defined constraints.

In accordance with other works [11, 21, 27] we use FSM to represent the protocol A. The FSM $A = (Q_A, \Sigma_A, \rightarrow_A, I_A, F_A)$ is defined as usual, i. e. Q_A is a finite set of states, Σ_A is a finite set of atomic actions, $\rightarrow_A \subseteq Q_A \times \Sigma_A \times Q_A$ is a finite set of transition rules, $I_A \in Q_A$ is the initial state, $F_A \subseteq Q_A$ is the set of final states.

Note that the protocol as FSM gives us the ability to show this protocol to the developers, to create a graphical representation (to use it in the development process), and to use it for automated verification.

Many approaches [11, 21] model the use of required interfaces by regular languages obtained by finite transducers. In [27] it is shown that this approach leads to false positives if recursion is present.

The use of a component C_i in a component-based system is the set of possible sequences of calls to C_i. Thus, this can be also modeled as a language L^i_{Π}. Hence the protocol conformance checking is equivalent to check whether $L^i_{\Pi} \subseteq L_{P_i}$, when L_{P_i} is the language defined by the protocol P_i of C_i.

In Figure 2 an example of a component-based system including the components implementations and protocols is shown.

[2] 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.

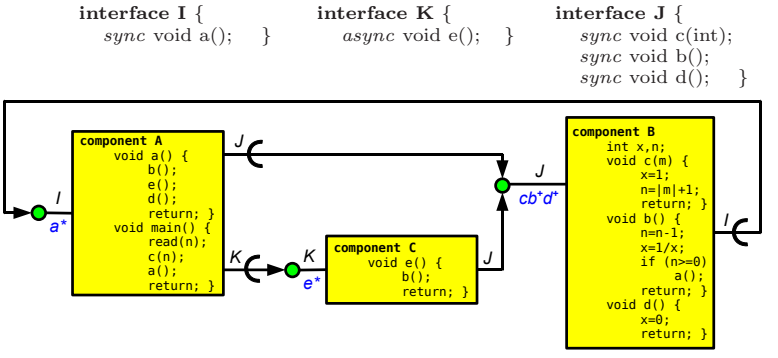

Fig. 2. Example. A component-based system assembled from the components A (implements interface I), B (implements interfaces J) and C (implements interface K). The components have following protocols, given as regular expessions: A: a^*, B: cb^+d^+, C: e^*.

In this work we will verify if components interfaces are used in the manner the developer specified by a protocol. By doing this, we can exclude semantic errors which appear because of an unexpected sequence of (remote) method calls.

For a more detailed proof we also need an abstraction of the behaviour of the component.

2.3 Process Rewrite Systems (Short PRS)

The abstracted behaviour of a component can be modeled with different representations. An abstraction $\mathcal{A}_C$ of a component C describes the behaviour $\mathcal{B}_C$ of C. Every possible execution path of C has a counterpart (trace) in $\mathcal{A}_C$. There exists a mapping from $\mathcal{B}_C$ to $\mathcal{A}_C$.[3]

An abstraction $\mathcal{A}_C$ has to implement every path of a component C, every control flowpath has to be recognized. In the work [27] parameterized context free systems (equal to PDA) were used to integrate recursion. The parameterization is required to implement callbacks. Because we transform a turing-powerful implementation to a not turing-powerful representation, the created abstraction $\mathcal{A}_C$ will approximate the behaviour of C, but this way we find a protocol violation, if there exists one.

As mentioned above, both representations (Petri nets and PDA) have advantages. We consider a representation which contains parallel semantic (like Petri nets) as well as sequential semantic (like PDA). Hence the base of this work will be the use of a representation called process rewrite systems (short PRS) defined by Mayr [16].

PRS unify the semantic of Petri nets and PDA. Mayr introduced an operator for parallel composition "||" and sequential composition ".". A process rewrite system $\Pi = (Q, \Sigma, I, \rightarrow, F)$ is defined as followed:

[3] The mapping $\mathcal{B} \rightarrow \mathcal{A}$ is often not bidirectional.

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.

We introduce a special action λ, denoting *no action* or *emtpy word*. The set $PEX(Q)$ contains all process-algebraic expressions over the set of atomic processes Q. The process rewrite rules define a derivation relation $\stackrel{a}{\Rightarrow} \in PEX(Q) \times \Sigma^* \times PEX(Q)$ by the following inference rules ($a \in \Sigma \cup \{\lambda\}, x \in \Sigma^*$):

$$\frac{(u \stackrel{a}{\rightarrow} v) \in \Pi}{u \stackrel{a}{\Rightarrow} v}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{u.w \stackrel{a}{\Rightarrow} v.w}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{u||w \stackrel{a}{\Rightarrow} v||w}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{w||u \stackrel{a}{\Rightarrow} w||v}, \quad \frac{u \stackrel{x}{\Rightarrow} v \;\; v \stackrel{a}{\Rightarrow} w}{u \stackrel{x\,a}{\Longrightarrow} w}$$

$L_\Pi \hat{=} \{w : \exists f \in F | I \stackrel{w}{\Rightarrow} f\}$ is the language accepted by Π.

Based on the operators Mayr defined a hierarchy of PRS classes that allows us the classification of process rewrite systems by the appearance of operands. Mayr uses the following base classes:

- 1: terms are composed of atomic processes only
- P: terms are composed of atomic processes or parallel composition
- S: terms are composed of atomic processes or sequential composition
- G: terms can be formed with all operators

These classes model different behaviour. Hence it is not possible to model all behaviour of a parallel system only with sequential composition and vice versa (cf. Figure 3a).

With the four base classes, a hierarchy based on bisimulation was formed (cf. Figure 3b), which allows us to classify all possible and sensible PRS.[4]

As we see, the $(1, S)$-PRS allows rules, which contain a process constant at the left-hand side and allows the sequential operator at the right-hand side. Thus this class is equivalent to PDA with one state, which accepts a language, if the stack is empty. The empty stack is represented in a $(1, S)$-PRS with the emtpy process ε. The (S, S)-PRS is the companion piece to PDA with several states.

PRS which allows the parallel operator are among others the class (P, P)-PRS which is equivalent to the well known Petri nets. The $(1, P)$-PRS are Petri nets, where every transition of the net has only one incoming arc, hence it does not contain synchronization.

By looking at the PRS hierarchy in Figure 3b we see, that if we search for a fusion of Petri nets with sequential concepts we have to use the (P, G)-PRS "PAN"[5], but also the $(1, G)$-PRS "Process Algebra" (short PA) could be interesting, if we have no synchronization of components.

[4] [16] points out that the left-hand side of a PRS-rule must be at most as large as the right-hand side in the sense of Figure 3a.

[5] Caused by the fact, that the grammar described by a (S, S)-PRS PDA can be accepted by a $(1, S)$-PRS called BPA, which is a pushdown automata with only one state.

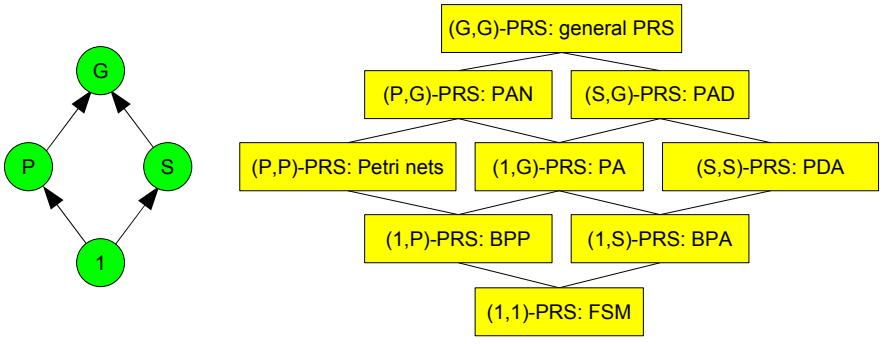

Fig. 3. a) Hierarchy of basic PRS operators, b) Hierarchy of process rewrite systems (cf. [2]), classification by appearance of operands on the left-hand side and right-hand side, (lhs,rhs)-PRS

Using PRS as an abstraction model, we are able to deal with real programs, because all important behaviour like recursion, threads, synchronous and asynchronous remote function calls can be modeled.

In this paper we will focus on $(1, G)$-PRS called Process Algebras because it can handle recursion and parallel behaviour, and our component model does not contain synchronization. Mayr has shown, that reachability is solvable, therefore Process Algebras can be considered for our application.

3 Building the Use of Components as PRS

Now we will create an PRS abstraction $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S)$ of the component-based System S in a $(1, G)$-PRS representation. We assume here, that the full source code is available. The main ideas for the construction are:

1. Create an atomic process for each program point p_i of a component C: Without loss of generality we assume, that every control flow path of a method ends with a `return`-statement. For `return`-statements no program point will be created.
2. Create transition rules, which map the control flow of the component in process rewrite rules: We use the mapping function $next : p_i \rightarrow p_j$, which results in the program points $p_j \in Q$, which are the possible succeeding program points of p_i. The mapping result contains ε if there exists a control path, that ends in the next step (`return`-statement).
   - If at a program point p_i a synchronous method call a is performed, we create rewrite rules $p_i \overset{a}{\rightarrow}_S p_j \ . \ p_k$ and $p_i \overset{a}{\rightarrow}_S p_j \ || \ p_k$, if a is an asynchronous method call.
   - If at a program point p_i another operation is performed, we create rewrite rules $p_i \overset{\lambda}{\rightarrow} p_k$, where $p_k \in next(p_i)$. This transition rule has the semantic, that this operation is not interesting for the protocol verification.

We always update Q_C, if we create a new rewrite rule.

Note that all these pieces of information can be derived automatically from the source code of the component and the interfaces used by the component. We have chosen a left-to-right evaluation order according to semantics of Java or C#. If the evaluation order of the regarded component is implementation-dependent one has to choose here the order used by a compiler.

Remark: As in [27], we can encode reference parameters in our component abstraction too, to regard even recursive call-backs. Also resolving the reference parameters to all possible dynamically chosen services is possible and equal to the mentioned earlier work. Because of the lack of space we do not describe these calculations here.

For technical reasons we add a new start rule $I_S \xrightarrow{\lambda} p_i$, where p_i is the first program point to be executed.

After this construction we get a Process Algebra, but it contains all possible remote method calls, so we have to eliminate every action which is not included in the protocol P_i of the component C_i that is checked. For this purpose we use the following mapping function Φ_i:

$$\Phi_i : \Sigma \to \Sigma_{C_i} \text{ defined by } \Phi_i(x) = \begin{cases} x & \text{if } x \in \Sigma_{C_i} \\ \lambda & \text{otherwise} \end{cases}$$

Where Σ_{C_i} containts all remote methods of component C_i. Thus every translation rule using an action x which is not part of the components protocol alphabet will be replaced by the same rule which uses λ as action. Now we have a representation $\Pi_S^i = (Q_S, \Phi_i(\Sigma), I_S, \to_S, F_S)$ of the component-based system S according to the component C_i, it is valid $L_{\Pi_S^i} \subseteq L_{P_i}$. Π_S^i is used to create the Combined Abstraction in the next section.

In Figure 4 the example[6] – mentioned before – has been extended by labelled program points. For better understanding, we chose these labels unique over all components. In Figure 5 we show the abstraction Π_S^B of the example component-based system according to the protocol P_B of the component B. Because our example has a main component, we can use the first directive to create a start rule of S, hence we create one extra start rule only.

4 Combined Abstraction

To verify the component-based system abstraction with respect to a component C_i, we have to check, if $L_{\Pi_S^i} \subseteq L_{P_i}$, where L_{P_i} is the regular language described by the components protocol P_{C_i}, and $L_{\Pi_S^i}$ is the language over the actions in Π_S^i, specifying a superset of the use of C_i. In order to check $L_{\Pi_S^i} \subseteq L_{P_i}$ it is usual

[6] Because of the lack of space the example has no reference parameters nor dynamically chosen services. All components are hard coded.

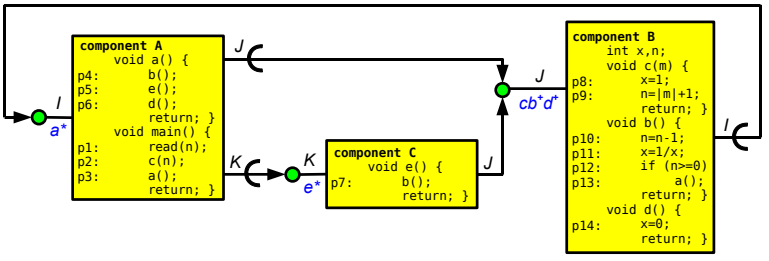

Fig. 4. Example. System with labelled program points.

$$I_S \xrightarrow{\lambda}_S p_1 \qquad p_4 \xrightarrow{b}_S p_{10}.p_5 \qquad p_8 \xrightarrow{\lambda}_S p_9 \qquad p_{12} \xrightarrow{\lambda}_S p_{13}$$

$$p_1 \xrightarrow{\lambda}_S p_2 \qquad p_5 \xrightarrow{\lambda}_S p_7||p_6 \qquad p_9 \xrightarrow{\lambda}_S \varepsilon \qquad p_{12} \xrightarrow{\lambda}_S \varepsilon$$

$$p_2 \xrightarrow{c}_S p_8.p_3 \qquad p_6 \xrightarrow{d}_S p_{14} \qquad p_{10} \xrightarrow{\lambda}_S p_{11} \qquad p_{13} \xrightarrow{\lambda}_S p_4$$

$$p_3 \xrightarrow{\lambda}_S p_4 \qquad p_7 \xrightarrow{b}_S p_{10} \qquad p_{11} \xrightarrow{\lambda}_S p_{12} \qquad p_{14} \xrightarrow{\lambda}_S \varepsilon$$

Fig. 5. Rewrite rules of Abstraction Π_S^B of the example component-based system according to the protocol of the component B

to check the equivalent problem $L_{\Pi_S^i} \cap \overline{L}_{P_i} = \emptyset$. Unfortunately this question is undecidable.

Theorem 1 (Undecidability of Protocol Checking Problem). *It is undecidable if $L_\Pi \subseteq L_P$ where Π is a $(1, G)$-PRS and L_P is regular.*

Proof (Sketch). As we know from model checkers (e. g. SPIN, MOPED), for each propositional LTL-formula ϕ, a FSM P can efficiently be constructed s. t. $L(\phi) = L_P$, where $L(\phi)$ is the set of action sequences specified by ϕ. Thus if the protocol checking problem would be decidable, we could also decide LTL-formula model checking for $(1, G)$-PRS. Contradiction, because LTL is undecidable in $(1, G)$-PRS. [5] □

We therefore construct a $(1, G)$-PRS K which describes a language $L_\approx$, where $L_{\Pi_S^i} \cap \overline{L}_{P_i} \subseteq L_\approx$. We call K *Combined Abstraction*. Thus if $L_\approx = \emptyset$, we know that $L_{\Pi_S^i} \subseteq L_{P_i}$. However, there might be a sequence $w \in L_\approx$ s. t. $w \notin L_{\Pi_S^i} \cap \overline{L}_{P_i}$. We call these sequences spurious false negatives.

In the following, we present the construction of the Combined Abstraction.

Roughly spoken the Combined Abstraction encodes in one model K the parallel execution paths of the abstraction of our system Π_S^i and the execution paths (which are forbidden by the regarded protocol P_i of C_i) formulated as finite state machine $\overline{P}_i$.

A combination of a protocol (FSM) A and an abstraction (PA) Π_S is to our knowledge not defined yet. The Combined Abstraction $K = (Q_K, \Sigma_K, I_K, \rightarrow_K, F_K)$ is defined as follows:

$Q_K = Q_A \times Q_S \times Q_A$	is a finite set of processes,
Σ_K	is a finite set of atomic actions,
$I_K \in Q_K$	is a start process,
$\rightarrow_K \subseteq Q_K \times \Sigma \times Q_K$	is a finite set of transition rules,
$F_K \subseteq Q_K$	is a finite set of final processes.

In accordance with [13] the processes $(q_i, q_j, q_k) \in Q_K$ encodes, that the FSM A is in the state q_i while the PA S has the process q_j created. The aim of (q_i, q_j, q_k) is, that $q_i \stackrel{x}{\Rightarrow}_A q_k$ while $q_j \stackrel{x}{\Rightarrow} \varepsilon$ can be performed.

The transition rules of the Combined Abstraction K have the same form and semantic as transition rules of PRS.

The construction of the other transition rules follows the directives shown in Figure 6 (described below) and is a generalization of the standard construction of the intersection of a finite state machine and a pushdown automaton in [13].

For technical reasons we also have to introduce the following two sets of rules:

$$R_S = \{I_K \stackrel{\lambda}{\rightarrow} (I_A, I_S, q_{F_A}) : q_{F_A} \in F_A\}$$
$$R_E = \{(q_A, q_P, q_A) \stackrel{\lambda}{\rightarrow} \varepsilon : q_A \in Q_A, q_P \in F_P\}$$

The chain transition rules R_{1C} and the set of sequential transition rules R_{1S} are handled similar to creating an intersection of pushdown automata and finite state machines in [13] (cf. Figure 6). The transition rules with a parallel operator R_{1P} are constructed as shown in directive r1p. As we see, a transition rule in the Combined Abstraction is similar to a transition rule in $\rightarrow_P$.

If one of the parallel threads in K accepts a $a \in \Sigma$ the protocol state in the other to p parallel threads should change to the same protocol states. With the transition rules in R_0, it is possible to implement these state changes.

The set of transition rules $\rightarrow_K$ is formed by uniting the sets R_S, R_E, R_{1C}, R_{1S}, R_{1P} and R_0.

After constructing the transition rules of the Combined Abstraction, we receive a rewrite system K in the syntax of PRS (we can also call K interleaving PRS). Now every possible interleaving sequence of the actions contained in the protocol is represented by at least one path in the Combined Abstraction K.

Theorem 2 (Correctness of construction of Combined Abstraction). *The construction K results in a representation, s. t. $L_{P_i} \cap L_{\Pi^i_S} \subseteq L_K$.*

Proof (Idea). Counterexamples constructable only by sequential rules are gathered by using the rewrite rules of R_{1C} and R_{1S}.

If a counterexample is conductable while using parallel rules, we have to look at rules of the form $(p_1||p_2).p_3$. Using the construction directive r1p the parallel traces are calculated independently.

If a rule out of R_{1P} is applied to p_1 thus this trace reaches a new state x in the protocol automaton. However in p_2 the protocol automaton has still the old

$$R_{1C} = \{(s,r,t) \xrightarrow{a}_K (s',r',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r') \wedge (a = \lambda)\} \quad \text{(r1c)}$$

$$R_{1S} = \{(s,r,t) \xrightarrow{a}_K (s',r',s'').(s'',r'',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r'.r'') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r'.r'') \wedge (a = \lambda)\} \quad \text{(r1s)}$$

$$R_{1P} = \{(s,r,t) \xrightarrow{a}_K (s',r',t)||(s',r'',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r'||r'') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r'||r'') \wedge (a = \lambda)\} \quad \text{(r1p)}$$

$$R_0 = \{(s,r,t) \xrightarrow{\lambda}_K (s',r,t) \quad :(s \xrightarrow{a}_A s')\} \quad \text{(r0)}$$

$$\text{with } s,s',s'',t \in Q_A;\ r,r',r'' \in Q_P;\ a \in \Sigma_A \cup \{\lambda\};$$
$$(s,r,t),(s',r',t),(s',r'',t),(s',r',s''),(s'',r'',t) \in Q_K$$

Fig. 6. Directives for construction of transition rules of a Combined Abstractions K

state. With a rule from R_0, it is possible that p_2 reaches x, too, while using a rule out of R_0.

As we can easily see, using the explained construction, we create false negatives, too. These will be described at the end of the next chapter.

5 Performing Protocol Checking

Now, we want to verify if there exists a path from the start process I_K to the empty process ε. This is a reachability problem.

As seen before we create $\overline{A} = (\Sigma_A, Q_A, \delta_{\overline{A}}, q_A, F_A)$, where $L(\overline{A}) = \overline{L}_A = \Sigma^* \setminus L(A)$. $\overline{A}$ contains all possible traces to the final states $q_{F_A} \in F_A$ which are not part of the protocol A, we want to verify. Using $\overline{A}$ and P, the Combined Abstraction K will be created as described in the previous section. After this construction every existing path $I_K \overset{*}{\Rightarrow} \varepsilon$ is a candidate for a counterexample, because ε encodes the error process. Creating the counterexamples is possible using the logic EF, which is decidable in the class of $(1,G)$-PRS. [17]

We get a sequence of actions s as counterexample. Because we named the process constants of the system abstraction as program points, we are able to point out each program point of the regarded component, where a possible protocol violation can appear.

If we look at the Combined Abstractions in Figure 5 we can easily see that there is no protocol violation in component A and C. But there is a protocol violation in B. In Figure 7 a trace constructed by the reachability algorithms is shown. For the lack of space we reduce the language $\overline{L}_B$ to the one given in Figure 7 and show only a trace which results in the counterexample *cbdb*. This counterexample can appear only, if the method call of e is asynchronously performed. If we look at the original source code of the component B in Figure 4 we can see, that this sequence of actions really results in a division by zero, which is a non expected behaviour.

FSM $P_{B'}$, that describes a subset of the inverted protocol $P_{\overline{B}}$ of component B.
$P_{B'} = (\{I_A, x_2, x_3, x_4, x_F\}, \{b, c, d\}, I_A, \{I_A \xrightarrow{c} x_2, x_2 \xrightarrow{b} x_3, x_3 \xrightarrow{d} x_4, x_4 \xrightarrow{b} x_F\}, \{x_F\})$ We can see that $L_{P_{B'}} \subseteq \overline{L}_B$.
Trace of K constructing the protocol violation *cbdb* in component B:

$\underline{(I_A, I_S, x_F)}$ $\stackrel{\lambda}{\Rightarrow} \underline{(I_A, q_1, x_F)}$ $\stackrel{\lambda}{\Rightarrow} \underline{(I_A, p_2, x_F)}$
$\stackrel{c}{\Rightarrow} \underline{(x_2, p_8, x_2)}.(x_2, p_3, x_F)$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_2, p_9, x_2)}.(x_2, p_3, x_F)$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_2, \varepsilon, x_2)}.(x_2, p_3, x_F)$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_2, p_3, x_F)}$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_2, p_4, x_F)}$ $\stackrel{b}{\Rightarrow} \underline{(x_3, p_{10}, x_3)}.(x_3, p_5, x_F)$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_3, p_{11}, x_3)}.(x_3, p_5, x_F)$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_3, p_{12}, x_3)}.(x_3, p_5, x_F)$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_3, \varepsilon, x_3)}.(x_3, p_5, x_F)$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_3, p_5, x_F)}$ $\stackrel{\lambda}{\Rightarrow} (x_3, p_7, x_F)||\underline{(x_3, p_6, x_F)}$ $\stackrel{d}{\Rightarrow} (x_3, p_7, x_F)||\underline{(x_4, p_{14}, x_F)}$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_3, p_7, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_4, p_7, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{b}{\Rightarrow} \underline{(x_F, p_{10}, x_F)}||(x_4, \varepsilon, x_F)$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_F, p_{11}, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{\lambda}{\Rightarrow} (x_F, p_{12}, x_F)||\underline{(x_4, \varepsilon, x_F)}$ $\stackrel{\lambda}{\Rightarrow} (x_F, p_{12}, x_F)||\underline{(x_F, \varepsilon, x_F)}$
$\stackrel{\lambda}{\Rightarrow} \underline{(x_F, p_{12}, x_F)}$ $\stackrel{\lambda}{\Rightarrow} \underline{(x_F, \varepsilon, x_F)}$ $\square$

Fig. 7. Example. Trace that will be constructed by the reachability algorithm, it results in the counterexample *cbdb* for the protocol of B in Figure 4. For better understanding the processes of K rewritten in the considered step are underlined.

False Negatives There are two causes of false negatives:

- real false negatives: Because the component abstractions are created without any data flow or control flow analysis, it is possible that a trace will be contained in the component abstraction, which is not possible in the implemented component. Moreover e. g. a return value can route the control flow, thus if we have no access to the implementation of the other components of the component-based system, it is possible to create more false negatives. The constructable counterexample *c* in our example is such a false negative.
- spurious false negatives: Because we only construct an approximated intersection of the language described by the component-based system and the regarded protocol, it is possible to get false negatives.

If the component code is not available, it will only be possible to reduce the spurious false negatives.

6 Component Composablity

We now show how a kind of PRS can be individually computed for each component, and how these can be composed in order to obtain a PRS describing the use of the component whose protocol has to be checked.

As in reality not every component (e. g. Web Service) is accessible by a component developer. Thus it is necessary to define an abstraction for each component C, which is composable to an abstraction of the full component-based

system. We call the PRS of a single component C *stripped process rewrite system* $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C)$. Π_C is defined as follows:

Q_C	is a finite set of atomic processes,
Σ_C	is a finite set of atomic actions,
$\rightarrow_C \subseteq PEX(Q_C) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q_C)$	is a set of process rewrite rules,
R_C	is a finite set of required interfaces,
$P_C : S \mapsto Q_C$	is a mapping from the services to the first program point in the provided interfaces.

The foundation for creating $\rightarrow_C$ are the directives described in Section 3. We extend the considered directives by the following:

- If at a program point p_i a synchronous *remote* method call a is performed, we create rewrite rules $p_i \xrightarrow{a}_C q_{J,s} \,.\, p_k$, if a is an asynchronous *remote* method call we create rewrite rules $p_i \xrightarrow{a}_C q_{J,s} \,||\, p_k$, where $q_{J,s}$ specifies the interface J of the required service s, and $p_k \in next(p_i)$. Note if we do not know how the interface is implemented, we have to create both sets of rewrite rules to ensure, that we create a conservative abstraction.
- If the considered program point p_i is the first in a method implementing a provided service s, we will extend the mapping P_C with $s \mapsto p_i$.

The set R_C contains all interfaces $q_{J,s}$ where s is a service of a required interface J. P_C mappes the set of services S (provided by the interfaces of C) to the initial process of the provided interface.

In the case considered in this paper, we have to look at stripped Process Algebras only. You can see the abstractions of the example components in Figure 8.

After having constructed abstractions for each component, we have to combine each component C_i (respectively Π_{C_i}) to the component-based system S (respectively Π_S) we want to verify. In the first phase, this can easily be constructed by uniting the relevant sets. Thus we define the abstraction $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S)$ of S as follows:

$Q_S = \{I_S\} \uplus \bigcup_{C_i} Q_{C_i}$	is a finite set of processes,
$\Sigma_S = \{\lambda\} \cup \bigcup_{C_i} \Sigma_{C_i}$	is a finite set of actions,
$I_S \in Q_S$	is a new start process,
$\rightarrow_S = \bigcup_{C_i} \rightarrow_{C_i} \cup Init$	is a finite set of transition rules,
$F_S = \bigcup_{C_i} F_{C_i} \subseteq Q_S$	is a finite set of final processes.

$$\rightarrow_A = \{p_1 \xrightarrow{\lambda} p_2,\ p_2 \xrightarrow{c} q_{J,c} \,.\, p_3,\ p_3 \xrightarrow{a} p_4,\ p_4 \xrightarrow{b} q_{J,b} \,.\, p_5,\ p_5 \xrightarrow{e} q_{K,e} \,||\, p_6,\ p_6 \xrightarrow{d} q_{J,d}\}$$

$$\rightarrow_C = \{p_7 \xrightarrow{b} q_{J,b}\}$$

$$\rightarrow_B = \{p_8 \xrightarrow{\lambda} p_9, p_9 \xrightarrow{\lambda} \varepsilon, p_{10} \xrightarrow{\lambda} p_{11}, p_{11} \xrightarrow{\lambda} p_{12}, p_{12} \xrightarrow{\lambda} p_{13}, p_{12} \xrightarrow{\lambda} \varepsilon, p_{13} \xrightarrow{a} q_{I,a}, p_{14} \xrightarrow{\lambda} \varepsilon\}$$

$$P_A(q_{I,a}) \mapsto p_4,\ P_C(q_{K,e}) \mapsto p_7,\ P_B(q_{J,c}) \mapsto p_8,\ P_B(q_{J,b}) \mapsto p_{10},\ P_B(q_{J,d}) \mapsto p_{14}$$

Fig. 8. Example. Transitions rules of the abstractions Π_A, Π_B, Π_C and mapping of provided interfaces of components A, B, and C in Figure 4.

To ensure that every initial program point I_i of a component C_i is reachable from the new start process I_S, we add the following transition rules to $\rightarrow_S$:

$$(I_S \xrightarrow{\lambda}_S I_i) \quad \text{iff } C_i \text{ is the main component/the client of } S$$

$$(I_S \xrightarrow{\lambda}_S I_i) \quad \text{iff any component } C_i \text{ of } S \text{ can start}$$

$$(I_S \xrightarrow{\lambda}_S I_0||I_1||...||I_n) \quad \text{iff each component } C_i \text{ of } S \text{ can process independently}$$

In the second phase, every used interface has to be resolved to a process, that specifies the first program point of the called interface implementation. Thus we have to resolve all interfaces $q_{J,s} \in \bigcup_{C_i} R_{C_i}$ using the mapping function P_{C_i} of the component implementing the interface J.

As in Section 3 we still have to create new start rules and eliminate every action, which is not included in the protocol P_{C_i} of the considered component C_i using the mapping function Φ_i.

In Figure 8 the abstractions of the components A, B and C are shown. As can easily be seen the abstraction S of the full system in our example is easy to build, unifying the sets and resolving the provided interfaces as mentioned above. It results in the PRS shown in Figure 5.

7 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 [18]. Their approach uses regular languages to model the dynamic behaviour of objects, which is less powerful than context-free grammars (CFG). In the work of Yellin and Strom [25] also only regular representations of the components are used, but they describe a protocol by send and receive synchronous method calls, and generate adapters if the protocol check fails. These approaches cannot handle recursive call-backs. [15] considers object-life cycles for the dynamic exchange of implementations of classes and methods using a combination of the bridge/strategy pattern. It also based on FSMs. The approach comprises dynamic as well as static conformance checking. Tenzer and Stevens [23] investigate approaches for checking object-life cycles.

They assume that object-life cycles of UML-classes are described using UML state-charts 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 and therefore could accept general context-free languages. All these works are for sequential systems. Schmidt et al. [12] propose an approach for protocol checking of concurrent component-based systems. Their approach is also FSM-based and unable to deal with recursive call-backs.

Even modeling the use of a component with context-free languages may abstract too much from the real behaviour. Other approaches [9, 20] therefore use dynamic protocol-checking. Dynamic protocol checking does not exclude protocol faults as static protocol checking does. On the other hand, they identify bugs at the right place. In particular, dynamic adapters might support avoiding protocol faults whenever possible.

An alternative approach for investigation of protocol conformance is the use of process algebras such as CSP, cf. e. g. [1]. These approaches are more powerful than FSMs and context-free grammars. 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 reduces to checking FSMs similar to [12].

FSMs are also used for checking Liskov's substitution principle for subtyping in object-oriented systems based on class protocols. Reussner [21] generalizes on the idea of Nierstrasz and adds counters and conditions over counters to the regular types to decide, whether Liskov's substitution principle is satisfied. Freudig et al. [11] use sub-classes of CFGs for describing protocols and checking Liskov's substitution principle. They need subclasses of CFGs because the subset-problem on general context-free languages is algorithmically undecidable. They do not model calling sequences stemming from a method which is required for checking whether the use of an object of a certain class conforms to its protocol.

The work on model checking context-free processes and pushdown systems started with [6, 7]. The model checking of LTL-formulas can be done linear in the size of the system and cubic in the number of states [2, 3, 10]. However, these approaches would require that the complete system is available as a context-free process or as a pushdown system. The framework described in [4] contains among others an algorithm for checking whether $L(G) \subseteq L(A)$ for context-free grammars G and finite state machines A.

The approach in this paper is a generalization of [26, 27]. In these papers recursion is modeled by CFG, so only sequential behaviour is considered. It is demonstrated how the approach can be made compositional. Moreover recursive callbacks are respected, which is possible but not considered within our work. Like in our approach every components abstraction has to be known at the verification time, but in contrast to this work counterexamples can be created exactly, if a fault has been discovered.

Chaki et al. described in [8] a method to verify communicating recursive C programs. This problem seems to be similar to verification of component-based systems, although they considered synchronous method calls only. In contrast to our work they consider even the data manipulation and synchronization statements. The problem can be reduced to the intersection of – by C programs described – context free languages, which were calculated approximately by a CEGAR-loop. There are other works [14, 19, 24] which consider the verification of concurrent programs, but these reduce the problem with bounded context switching, which results in a bounded parallelism.

8 Summary and Conclusions

In this paper we discussed the automatic verification of components according to their protocol. This static verification can be used to find semantic errors, i. e. to verify defined non functional business rules.

To apply our check, we require static knowledge of the used components. But this abstraction can be part of the component description, so we do not need access to the components source code. As other works in this research area, we use FSM for describing component protocols.

In contrast to previous approaches, we are able to handle recursion and parallel behaviour in a local and global view without any restrictions using process rewrite systems to represent the behaviour of each component instead of finite state machines or context free grammars. The decidability of the reachability problem has been proven by Mayr. In order to circumvent the undecidability of the protocol checking problem, we define an approximated intersection of protocols and Process Algebras – the so called Combined Abstraction.

We implemented a two phase process to consider the component composition, where in the first phase the components were composed, like in the real system and in the second phase the required interfaces (and reference parameters) were resolved, so every information depending on the component-based system can be included in the system abstraction. Because of this process we are able to compose the abstractions of the components like the components in reality. Moreover our approach makes it possible to deal with components implemented in different programming languages, because the abstraction layer hides the implementing details.

The tool provides a counterexample if the protocol conformance check fails. So our approach is a model-checking approach. A counterexample is a word over all protocol actions, which are remote method calls. A calculated counterexample may not occur in the real system, because we create a conservative abstraction, hence false negatives may be delivered. But we are sure to find a counterexample if any exists.

At this stage of our work we only consider static verification, i. e., the abstractions of each component are known statically. CORBA, COM, .NET and EJBs also allow dynamic instances of components. It is subject to further work to handle this property. As demonstrated in [27], a points-to analysis might help to solve the problem.

Our approach is adaptable for object-oriented programming where the protocols are defined over the public interfaces. It will be part of future work to research if our approach is suitability for daily use in OOP.

We currently implement a framework which creates abstractions of components implemented in Python (finished) and C++ (in progress). Creating abstractions of Java components is planed. This framework is currently be used to verify our approach in industrial case studies. Early results show that our approach is applicable and can result to real (so far undiscovered) bugs.

False negatives may be reduced by integration of data and control flow analysis algorithms into the component abstraction process.

We thank Heinz W. Schmidt for pointing us to process rewrite systems.
We are grateful to OR Soft GmbH for providing us with industrial case studies.

References

1. Allen, R., Garlan, S.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
2. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 207–220. Springer, Heidelberg (2001)
3. Benedikt, M., Godefroid, P., Reps, T.: Model checking of unrestricted hierarchical state machines. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 652–666. Springer, Heidelberg (2001)
4. Bouajjani, A., Esparza, J., Finkel, A., Maler, O., Rossmanith, P., Willems, B., Wolper, P.: An efficient automata approach to some problems on context-free grammars. Information Processing Letters 74(5-6), 221–227 (2000)
5. Bouajjani, A., Habermehl, P.: Constrained properties, semilinear systems, and petri nets. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 481–497. Springer, Heidelberg (1996)
6. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
7. Burkart, O., Steffen, B.: Pushdown processes: Parallel composition and model checking. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 98–113. Springer, Heidelberg (1994)
8. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing c programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
9. Chambers, C.: Predicate classes. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 268–296. Springer, Heidelberg (1993)
10. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
11. Freudig, J., Löwe, W., Neumann, R., Trapp, M.: Subtyping of context-free classes. In: Proceedings 3rd White Object Oriented Nights (1998)
12. Schmidt, H.W., Krämer, B.J., Poernemo, I., Reussner, R.: Predictable component architectures using dependent finite state machines. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 310–324. Springer, Heidelberg (2004)

13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
14. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
15. Löwe, W., Neumann, R., Trapp, M., Zimmermann, W.: Robust dynamic exchange of implementation aspects. In: TOOLS 29 – Technology of Object-Oriented Languages and Systems, pp. 351–360. IEEE, Los Alamitos (1999)
16. Mayr, R.: Process rewrite systems. Information and Computation 156(1-2), 264–286 (2000)
17. Mayr, R.: Decidability of model checking with the temporal logic ef. Theor. Comput. Sci. 256(1-2), 31–62 (2001)
18. Nierstrasz, O.: Regular types for active objects. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, pp. 99–121. Prentice-Hall, Englewood Cliffs (1995)
19. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS, pp. 93–107 (2005)
20. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 83–94. ACM, New York (2002)
21. Reussner, R.H.: Counter-constraint finite state machines: A new model for resource-bounded component protocols. In: Grosky, W.I., Plášil, F. (eds.) SOFSEM 2002. LNCS, vol. 2540, pp. 20–40. Springer, Heidelberg (2002)
22. Schmidt, H.W., Krämer, B.J., Poernomo, I., Reussner, R.: Predictable component architectures using dependent finite state machines. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 310–324. Springer, Heidelberg (2004)
23. Tenzer, J., Stevens, P.: Modelling recursive calls with uml state diagrams. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 135–149. Springer, Heidelberg (2003)
24. Torre, S.L., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
25. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2), 292–333 (1997)
26. Zimmermann, W., Schaarschmidt, M.: Model checking of client-component conformance. In: 2nd Nordic Conference on Web-Services. Mathematical Modelling in Physics, Engineering and Cognitive Sciences, vol. 008, pp. 63–74 (2003)
27. Zimmermann, W., Schaarschmidt, M.: Automatic checking of component protocols in component-based systems. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 1–17. Springer, Heidelberg (2006)