Supporting the Development Process of Reliable Software During the Composition Process Using Interaction Protocols

Andreas Both, Wolf Zimmermann
Supporting the Development Process of Reliable Software During the Composition Process Using Interaction Protocols

Andreas Both, Wolf Zimmermann
Developing reliable software is an expensive task that needs more support by formal techniques. Many approaches focus on the verification of applications after their development. We use interaction protocols to ensure the functionality of black-box components in applications using source code abstractions. These abstractions can capture unbound recursive and unbound parallel behavior. Our previous approach considers a complete system after the construction (or composition).

In an industrial environment components are often developed concurrently by independent developers and bound loosely. Attaching the verification at the end of the verification process might result in higher development costs because problems are discovered late.

In order to avoid these costs it is necessary to integrate the verification into the development process of single components and provide pieces of information for the developers. In this paper, we present an iterative verification approach using interaction protocols as constraints. The approach guarantees that previous positive verification results remain valid while extending a system.

1 Motivation

Developing software contains nowadays a big share of reusing previously developed software called components (e.g., Web Services). Often components are composed to larger components, ensuring wished functionality, used in the further development, or sold for reusing as a part of unknown, future applications. Thus, there is a big interest in components which can be composed to reliable systems.

Using toolkits (to develop reliable software) is an important task and indispensable, while standing in strong competition with other companies. This is important especially while providing only components (e.g., Web Services) used in other applications (e.g., B2B components). In these fields of application it is important to provide reliable components because every failure (e.g., crash) of a component or problem can cause costs and loss of earnings. Moreover violating business rules can cause legal problems.

To prevent these problems we suggest in earlier works [BZ08b, BZ08a] a formal approach allowing to formulate constraints by using so called interaction protocols. An interaction protocol
is an easy way to formulate constraints which could be derived from business rules as well as from safety or implementation reasons. Interaction protocols make the workflow of a component transparent to users and usable for model checking.

To allow the usage in a multi component provider environment we use abstractions of every single component, which are composed to imitate the behavior of the full application. This is important because often the business secrets implemented in these components can not be disclosed. In our works we show how we can use Process Rewrite Systems (PRS) [May00] to capture conservatively the behavior of source code. PRS unify Petri nets and Pushdown Automata. Hence they can model recursion and parallelism without any restrictions. Moreover we defined in these works a verification process implementing the preparation and static verification of the interactions of programs. We check if the protocols of each component are obeyed within the constructed applications (protocol conformance).

From our point of view providing (only) a verification process which ensures the defined constraints is not enough. This will ensure the properties of the application, but using this verification process problems are discovered later. This might cause higher costs for adaptions. Moreover this approach is not suitable for verifying e.g., B2B scenarios or Service-oriented Architectures, where a complete application not always exists or the components are bound dynamically.

Thus, the verification process should be integrated into the development process and composition process. In this paper, we will show how the available pieces of information about the behavior and the constraints of components can be used to support the development/composition of components. This will lead to a permanent verifyability of the current development state without having the full application implemented.

We allow to partially compose components and verify this new aggregated component (also known as composite component). The verification of this aggregated component is also possible if no implementation of the required interfaces is available. For this purpose we generate verification drivers and verification dummies which are similar to the test driver and test dummy, during test driven software validation. In contrast to tests, we consider every possible state of the software and we can prove the absence of the considered kind of errors (protocol violations). The verification results in indications which implementations might hurt the defined interaction protocols or that the component is reliable. This will lead to a bigger chance to discover problems in an earlier phase, which will reduce the investments for changing the implementations or requirements and allow to apply the verification to more scopes of application.
interface I2 {
    void b(int);
    void d();
}

interface I1 {
    void a(int);
}

interface I3 {
    void e(int);
    void f();
}

Figure 1: Example: Application with three component, each with interaction protocol.

The paper is organized as follows. In the next section (Section 2) the foundations are discussed. This includes our component model in Section 2.1, the depiction of the interaction protocols in Section 2.2 and the representation of the source code behavior in Section 2.3. In Section 3 we will give a brief overview of the existing verification process working with complete applications assembled from components. In Section 4 we give an overview about the new approach to verify even aggregated components, hence components which are already bound but not executable. The details and proofs of this process are presented in Section 5 and 6. The paper finishes with a consideration of the related work in Section 7 and the conclusion and future work in Section 8.

2 Foundations

2.1 Component and Component Model

The components in this work are represented using a model similar to the UML component model. They have to implement provided interfaces (method they provide for the use by of other components). Moreover they have required interfaces (remote methods the component might call).
The provided interfaces are predefined in an interface description. Our component model is shown in Figure 2.

We assume here that it is known, whether a component call should be implemented blocking (synchronous, short: sync) or non-blocking (asynchronous, short: async). In [BZ08a] it is described how method calls are handled, if this is not known.

The invocation implementation is denoted at the interface definitions. Components may be implemented in any imperative or object-oriented programming language [BZ08b].

Figure 1b shows an example containing three components. Note that the example does not include parallel behavior since all interfaces are implemented synchronously.

The example implements an application that crashes in every case where the initial method main is called with a value greater than 0. Unfortunately this can only be discovered after the composition of the three components, because the error is triggered by a callback of b (of C₂) triggered by an execution of a (of C₁), that initiates a call to e (of C₃).

2.2 Interaction Protocols

An interaction protocol (short: protocol) describes the allowed use of all callable operations (cf. interfaces) of a component. We want to verify statically, if a component is always used in the manner specified by the protocol. Especially while reusing a component, it has to be ensured that the usage does not result in (functional or non-functional) faults. Protocols are used for example to avoid (uncaught) exceptions (like division by zero, an example is presented in [BZ08b]) during execution or to obey business rules (example in [BZ08a]).

We will check automatically, if an application (assembled from components) obeys the defined constraints (interaction protocol).

In accordance with other works [ZS06, Reu02, FLNT98] we use finite state machines (short: FSM) to represent the protocol \( P \). The FSM \( P = (Q_P, \Sigma_P, \rightarrow_P, I_P, F_P) \) is defined as usual, i.e., \( Q_P \) is a finite set of states, \( \Sigma_P \) is a finite set of atomic actions, \( \rightarrow_P \subseteq Q_P \times \Sigma_P \times Q_P \) is a finite set of transition rules, \( I_P \in Q_P \) is the initial state, \( F_P \subseteq Q_P \) is the set of final states. The FSM

\(^1\)Examples containing parallel behavior can be found in [BZ08b, BZ08a, BZ09].
A. Both and W. Zimmermann: Supporting the Development Process of Reliable Software

Figure 3: Example: process algebraic-expression and corresponding cactus stack.

```
function x() {
    p1: if ( ... ) then
    p2:    a(); // synchronous call
    p3: ...
    p4: b(); // asynchronous call
    p5: return; }
```

(a) Source code

```
\( p_1 \xrightarrow{\lambda} p_2 \)
\( p_1 \xrightarrow{\lambda} p_3 \)
\( p_2 \xrightarrow{a} r_a p_3 \)
\( p_3 \xrightarrow{\lambda} p_4 \)
\( p_4 \xrightarrow{b} r_b p_5 \)
\( p_5 \xrightarrow{\lambda} \varepsilon \)

(b) SPRS
```

Figure 4: Introduction: Component model and translation of source code to SPRS.

\( \mathcal{P} \) defines a regular language \( L(\mathcal{P}) \).

We assume that a protocol \( \mathcal{P}_i \) for each component \( C_i \) exists. If no protocol is defined (by human) we fix the protocol such that \( L(\mathcal{P}_i) = \Sigma_{C_i}^* \), where \( \Sigma_{C_i} \) is the set of callable operations of \( C_i \). Hence every interaction sequence with \( C_i \) is allowed by \( \mathcal{P}_i \).

The protocol of the component \( C_2 \) is shown in Figure 1c as regular expression\(^2\). The protocol \( \mathcal{P}_{C_2} \) only allows calls to component \( C_2 \) where after an interaction \( b \) a call to \( d \) is performed. Thus, the developer of \( C_2 \) has already discovered the problem resulting in a crash and has forbidden the particular interaction sequence. The formal definition of this protocol is: \( \mathcal{P}_{C_2} = (\{v_0, v_1\}, \{b, d\}, \{v_0 \xrightarrow{b} v_1, v_1 \xrightarrow{d} v_0, v_1 \xrightarrow{d} v_1\}, v_0, \{v_0\}) \).

**Remark 1** In contrast to other works (e.g., behavioral protocols in [PV02]), our protocol contains only the callable operations of a component not the behavior of a component. I.e., the abstracted behavior and the protocol are represented and published independently. Thus it allows more flexibility when exchanging the implementation of a component.

### 2.3 Abstractions

To capture the behavior (the sequences of interactions with other components) of a component we need an abstraction of the source code. We capture conservatively every controlflow path of the

\(^2\)Often we will use the regular expression representation because of its shortness.
considered component. Thus it is possible to find all existing errors with respect to a protocol.

The natural execution model for capturing unbound recursion and unbound concurrency represents states as cactus stacks. 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 saguro stack). Hence an execution transforms a cactus stack into cactus stack. Process-algebraic expressions can be used to represent cactus stacks. Let \( PEX(Q) \) be the set of process-algebraic expressions over a finite set \( Q \) of atomic processes and the binary operators \( \otimes \) (for sequential composition, associative) and \( \|\| \) (for parallel composition, commutative and associative), where a cactus stack naturally represents a process-algebraic expression \( t \in PEX(Q) \) (and vice versa).

In Figure 3 an example is presented showing a process-algebraic expression and the corresponding cactus.

We use Process Rewrite Systems (PRS) [May98] as representations of the behavior. They rewrite process-algebraic expressions into process-algebraic expression. Therefore their use to describe the semantics of concurrent systems with unbound recursion and unbound parallelism seems natural. Thus PRS can capture a component behavior even if recursion or (recursive) callbacks are present.\(^3\)

A \( \text{PRS} \Pi \equiv (Q, \Sigma, I, \rightarrow, F) \) where \( Q \) is a finite set of atomic processes, \( \Sigma \) is a finite alphabet over actions, \( I \in Q \) is the initial process, \( \rightarrow \subseteq PEX(Q) \times \Sigma \times PEX(Q) \) is a set of process rewrite rules, \( F \subseteq PEX(Q) \) is a finite set of final processes. The process \( \varepsilon \) denotes the empty process in this work. It is the identity on \( \|\| \) and the left identity on \( \otimes \).

The process rewrite rules define a derivation relation \( \Rightarrow \in PEX(Q) \times \Sigma^\ast \times PEX(Q) \) by the

\(^3\)In this case Petri nets are not suitable, because they can not capture recursion.
following inference rules ($a \in \Sigma, x \in \Sigma^*$):

\[
\begin{align*}
& t \xrightarrow{\lambda} t & (t_1 \xrightarrow{a} t_2) \in \Pi, & t_1 \xrightarrow{a} t_2, & t_1 \xrightarrow{a} t_2, & t_1 \xrightarrow{a} t_2, & t_1 \xrightarrow{a} t_2, & t_1 \xrightarrow{a} t_2, \\
& t \xrightarrow{a} t_2, & t_2 \xrightarrow{a} t_3, & t_3 \xrightarrow{a} t_3, & t_3 \xrightarrow{a} t_3, & t_3 \xrightarrow{a} t_3, & t_3 \xrightarrow{a} t_3, & t_3 \xrightarrow{a} t_3.
\end{align*}
\]

They unify the semantics of Petri nets and Pushdown Automata (PDA). We introduce a special action $\lambda$, denoting no relevant interaction or empty word.

$L(\Pi) = \{ w : \exists f \in F | I \xrightarrow{w} f \}$ is the language accepted by $\Pi$.

Mayr shows in [May98] that the rules of any PRS can be transformed to a normal form, i.e., the LHS and RHS has one of the forms $t_1, t_1.t_2$ or $t_1||t_2$, where $t_1$ and $t_2$ are atomic processes. We assume in this work that every PRS has been transformed to a PRS in normal form.

Hence, the behavior of the system assembled from components is represented using PRS. In Figure 4 we show an example of how the behavior of a source code can be captured using PRS.

In a component system the source code of the used component is often not accessible. Therefore we request that the behavior of a single component is translated into an abstraction. Abstractions of components should be as composite as the actual components, thus they contains the same required and provided interfaces as the actual component. We denote component abstractions as Stripped Process Rewrite System (SPRS). In contrast to the regular PRS a SPRS $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C, M_C)$ contains also a mapping $M_C : P_C \mapsto Q_C$ to program points $Q_C$ identifying the provided interfaces $P_C$ and a set $R_C$ helping identifying the calls to required interfaces. The mapping of the example (Figure 4) is $M = \{ q_x \mapsto p_1 \}$, because $p_1$ is the initial program point of the method $x$. Abstractions can be constructed using well known compiler construction techniques (however, simplified approaches can also be used to construct abstractions as we have shown in [BZ08a]).

The abstractions of the components in Figure 1b are shown in Figure 5.

To generate the abstraction of the complete application the SPRS (of all includable components) are unified and the required interfaces are resolved using the given mappings $M$. Thus the source code of the used components is not needed to generate the abstraction of the application assembled from components. For details of the abstraction process we refer to [BZ08b, BZ08a].

2.4 Protocol Conformance

To check if a given protocol $P_i$ is obeyed in the application $S$ we have to verify if $L(\Pi^i_S) \subseteq L(P_i)$ [BZ08b], where $\Pi^i_S$ is the adopted application abstraction $\Pi_S$. $\Pi^i_S$ is created by replacing all interactions of $\Pi_S$ by $\lambda$, that do not belong to the interfaces of $C_i$ by $\lambda$. Thus $\Pi^i_S$ still represents the behavior of the complete application, but only the interactions with the component $C_i$ are
Definition 1 (\(\Pi_S^i\) of \(\Pi_S\)) \(\Pi_S = (Q, \Sigma, I, \rightarrow, F),\) considering the actions of \(C_i\) \(\Pi_S^i = \varphi_{C_i}(\Pi_S) = (Q, \Sigma_{C_i}, I, \theta_{C_i}(-), F),\) where \(\theta_{C_i} : t \xrightarrow{x} t' \in \rightarrow \{ \lambda : x \notin \Sigma_{C_i} \}, t, t' \in PEX(Q)\) and \(\Sigma_{C_i}\) are the provided interfaces of \(C_i\).

3 Verification Process

The verification process (cf. Figure 6) creates an abstraction \(\Pi_S\) of the system behavior. Each possible interaction between components is represented by \(\Pi_S\). In Figure 5 we see the abstraction \(\Pi_S\) of the system \(S\) composed by the abstractions of the components \(C_1, C_2\) and \(C_3\) (cf. Figure 1). The composition is possible using the composition operator \(\oplus\), which resolves the required and matching provided interfaces of the integrated components.

Definition 2 (Composition operator \(\oplus\)) The result of the composition operator \(\oplus\) is defined by the SPRS \(\Pi = \Pi_1 \oplus \Pi_2\), where \(\Pi = (Q, \Sigma, \rightarrow, R, P, M), \Pi_1 = (Q_1, \Sigma_1, \rightarrow_1, R_1, P_1, M_1), \Pi_2 = (Q_2, \Sigma_2, \rightarrow_2, R_2, P_2, M_2)\) with \(Q = (Q_1 \cup Q_2) \backslash \{ r_x : q_x \in P_2 \}, \Sigma = \Sigma_1 \cup \Sigma_2, \rightarrow = (\rightarrow_1 \cup \rightarrow_2) \backslash \{ p \xrightarrow{\exists} p' \otimes p'' : p \xrightarrow{\exists} r_x \otimes p'' \in \rightarrow_1, M(r_x) = p', \otimes \in \{., ||\}, R = (R_1 \backslash \{ r_x : p_x \in P_2 \}) \cup R_2, P = P_1 \cup (P_2 \backslash \{ q_x : r_x \notin R_1 \}), M = M_1 \cup (M_2 \backslash \{ M(x) \mapsto p \in r_x \notin R_1 \}).\)

It is possible to bind the required to the own provided interfaces: \(\Pi = \Pi_{C_1} \oplus \Pi_{C_1}\).

Thus every execution path of the application \(S\) is captured by the abstraction \(\Pi_S\). Now, a SPRS \(\Pi_S^C\) is constructed such that \(L(\Pi_S^C) \supseteq L(P) \cap L(\Pi_S, C)\), i.e., \(\Pi_S^C\) contains all sequences of interactions that are forbidden by the protocol \(P\) of one component \(C\) but nevertheless exist in the program \(S\). It is not possible to construct a SPRS such that \(L(\Pi_S^C) = L(P) \cap L(\Pi_S, C)\)\(^4\). Thus we cannot verify in all cases whether a interaction sequence is contained in an application (abstraction \(\Pi_S, C\)), that is forbidden by the protocol \(P\). Since it is decidable whether \(L(\Pi) = \emptyset\) [May00] (where \(\Pi\) is a SPRS), we define an overapproximated of the intersection \(L(P) \cap L(\Pi_S, C)\): the Combined

\(^4\)Since it is undecidable whether \(L(\Pi_S, C) \subset L(P)\)
Abstraction $\Pi_S^C$. Thus every error can be found, because the Combined Abstraction $\Pi_S^C$ contains all paths from the initial program point to the final state describing an forbidden interaction sequence: $I \xrightarrow{\delta} \varepsilon$ [BZ08b].

Finally the $\Pi_S^C$ is model checked\footnote{Thus, the verification is performed statically, i.e., before the deployment of the application.}. The result contains counterexamples $\varepsilon$. These derivation paths $\varepsilon : I \xrightarrow{w} \varepsilon$ of $\Pi_S^C$ are equivalent to possible sequences of interactions $w$ violating the considered protocol. For further (technical) details we refer to [BZ08b, BZ08a]. In this work, we consider static verification only.

\section{Iterative Verification}

During the development process several scenarios exist concerning which components are already developed. The other components may not be available or in a state that can be used for verification. Figure 7 shows the architecture of some typically aggregated components $C_A$. We assume that there is only one internal component $C$ that provides the provided interfaces of the aggregated component $C_A$. We distinguish the following scenarios:

\begin{itemize}
\item[a)] $C_A$ has provided interfaces, but has no required interfaces (Figure 7a).
\item[b)] $C_A$ has provided interfaces, required interfaces exist which are still unbound, callbacks are not allowed (Figure 7b).
\item[c)] $C_A$ has provided interfaces, required interfaces exist which are still unbound, and arbitrary callbacks (including callbacks) are allowed (Figure 7c).
\end{itemize}
Scenarios a) and b) are important for the hierarchical composition of components. However, a component-based software is not always built up hierarchically. It is also possible that the components are composed in a flat manner, as is often done in Service-oriented Architectures. This means that every component may have required and provided interfaces. In this case protocol violations can be caused by (recursive) callbacks, too. These kinds of protocol violations are not easy to discover (above all by humans) and a local protocol check is insufficient [ZS06]. Thus scenario c) matters especially if callbacks can happen.

In Figure 7d a more complicated example is shown, which is also captured here, i.e., except the fact that the provided interface of an aggregated component is provided by a single internal component, there are no further restrictions on the architecture.

The example demonstrates that not every component required for composition is already implemented. For verification purpose, we need to know restrictions on the use of the aggregated component in a component-based software. We call these restrictions context. They play a similar role as the test driver and the test dummy for local tests.

As we can see in Figure 7a (component \(C_1\)) there may be components without required interfaces. We call these components base components. Usually bottom-up constructions of component-based software systems start with base components.

We assume that the integrated components \(C_i\) are composed to a new component, which we call aggregated component \(C_A\). The goal is to compose components \(C_i\) to a larger component \(C_A\), which can be handled easier. Thus required and provided interfaces of the components \(C_i\) are bound closely. The aggregated component can also have provided and required interfaces.

**Definition 3 (Architecture of the aggregated component)** An aggregated component \(C_A\) consist of at least two components, where the provided interfaces of components are bound to the required interfaces of other components (cf. Figure 7). We restrict the model here with the assumption, that only the provided interfaces of exactly one component \(C\) are unbound and thus used as provided interfaces of the aggregated component \(C_A\). Hence the protocol of \(C_A\) is equal to the protocol \(P_A\) of \(C\).

**Definition 4 (Top component)** A top component \(C\) is contained in an aggregated component \(C_A\), where the set of provided interfaces of \(C_A\) is equal to the set of provide interfaces of \(C\). Thus \(C\) implements the well known design pattern facade [GHJV95].

A context represents every possible implementation of a component which can be bound to interfaces of an aggregated component \(C_A\).
Definition 5 (Context) We distinguish three kinds of contexts:

- A verification driver $C^Q_A$ of $C_A$ has no provided interfaces, but for every provided interface $q_i$ of the aggregated component $C_A$ exists a matching required interface $r_i$ in $C^Q_A$.

- A verification dummy $C^R_A$ of $C_A$ has no required interfaces, but for every required interface $r_i$ of the aggregated component $C_A$ a matching provided interface $q_i$ in $C^Q_A$ exists. This context accepts all calls of the aggregated component $C_A$ to its required interfaces.

- A (complete) verification context $C^A_A$ of $C_A$ has the same required interfaces $r_i$ as the verification driver $C^Q_A$ and the same provided interfaces $q_i$ as the verification dummy $C^R_A$.

Figure 7 shows all different kinds of contexts.

An aggregated component $C_A$ may have provided interfaces but no required interfaces, cf. Figure 7a. In this case the protocols of all components of the aggregated component should be obeyed, if it has been proven that the aggregated component is reliable and $C_A$ is bound to other components.

Informally, to ensure that no protocol is violated, we create a context $C^Q_A$ of the aggregated component $C_A$ called verification driver (Section 5.1). It initiates all possible and legal calls to $C_A$. The model checker calculates the statement if the component $C_A$ is ready for the integration with other components (Section 4.2).

For the same reasons, we define the verification dummy $C^R_A$. It has at its provided interfaces the required interfaces $R_{C_A}$ of $C_A$, and each initial state of a call $x$ is terminated immediately (because no callbacks are initiated).

If no callbacks are allowed, it is sufficient to have a verification driver as well as a verification dummy. However, if callbacks are present, a (direct or indirect) call to a required interface of the aggregated component $C_A$ may initiate a call to a provided interface of $C_A$. Thus, a (complete) verification context $C^A_A$ (Section 5.2) is required, cf. Figure 7c and 7d.

The contexts are used as a substitute for the unknown components. Hence the model checking approach described in Section 3 can be applied.

4.1 Verification Process for Iterative Development

As we have seen in Section 3 we need the abstractions of every single component which is needed for the application. Only these abstraction will ensure that the behavior of the defined interaction protocols are obeyed. If the behavior of a component is not present, it is currently not possible to
decide whether the protocol of the considered component is obeyed. We have shown in Section 3 how these required pieces of information can be generated conservatively. The resulting component abstraction can be used in the verification process.

The new verification process is shown in Figure 8. We divide the verification into two parts, where the first consider (hierarchical) programs without callbacks, while the second consider also callbacks. This will allow the users to consider only the problems they are interested in. Moreover this will improve the model checking speed [BZ09], because the system abstraction contains less rules than in a non-iterative approach checking approach.

4.2 Evaluating the Result of the Model Checker

From the model checker we will get the result if a component using the provided interfaces of $C$ and obeying the protocol $P_A$ can trigger errors (protocol violation) in the other components $C_i$ of the aggregated component. If a model checking results in a counterexample we can directly describe in which constellation the counterexample appears and the protocol $P_i$ of which component $C_i$ is violated.

If a counterexample which violates protocol $P_i$ (which is no false negative) we can react with the following options for reactions are possible:

1. It can be checked locally, whether the considered interaction protocol $P_i$ should be weakened.
   This is only possible if no fault in the program will result by this adaption.

2. The other components $C_j$ (where $i \neq j$) participating at the counterexample in order to decide whether an adaption of their source code should be recommended.
3. The error can be consciously neglected because of assuming that such an error will not occur in a real implementation $C_1$ using the provided interfaces of $C_A$.

The first and second option will result in a situation where no protocol violation can appear, if any user of the aggregated component $C_A$ obeys the protocol $\mathcal{P}_A$. The advantage of this extension is that unviolated protocols $\mathcal{P}_j$ of $C_A$ needs to be considered (again), because all calls to the component $C$ are guarded by the protocol $\mathcal{P}_A$. In Theorem 1 we will prove that this is sufficient for avoiding protocol violations of internal components.

As described this is only possible if no counterexample is constructed (or the first or second option is implemented). In the third case all protocols $\mathcal{P}_i$ of the components $C_i$ which are violated while using the verification context have to be considered.

The abstraction of the aggregated component can now be reused in the further development or composition process.

**Remark 2** The possibility not to consider protocols reducing the protocols checks during the next step of the iterative composition or development, respectively. Moreover all transition rules of the form $t_1 \xrightarrow{a} t_2$ can be translated into $t_1 \xrightarrow{\lambda} t_2$, where $a \in \Sigma_{\mathcal{P}_i}$ and $\mathcal{P}_i$ is not violated in this composition step. This enables more statical reductions of the number of transition rules, resulting in a faster model checking [BZ09].

We formalize and prove these assumptions and the verification process in the following sections.

## 5 Verification Contexts (No Callbacks Allowed)

We use the component term to describe the behavior of a context. If the context is (really) implemented as component $C$ we would generate a new abstraction $\Pi_C$ to use this in our verification process (cf. Section 3). Thus $C$ is translated to a SPRS $\Pi_C$. To represent the behavior of a context $C'$ we also use a SPRS $\Pi_{C'}$ to represent the allowed behavior. For technical reasons (which become clear later) we assume that each call to $C_A$ has at least one terminating computation (this only excluded the possibility that a call never terminates).

### 5.1 Components with no unbound required interfaces

In this section we consider the scenario $a$) (cf. Section 4) of Figure 7a (considering base components).
We only consider the case where the interface is implemented synchronously. The asynchronous A. Both and W. Zimmermann: Supporting the Development Process of Reliable Software

case is not considered.

Definition 6 (verification driver \( C_A^2 \)) A verification driver for \( C_A \) is a SPRS \( C_A^2 = (Q_{CA}, \Sigma_{CA}^2, \rightarrow_{CA}^2, R_{CA}^2, F_{CA}^2, M_{CA}^2) \) where:

\[
\begin{align*}
Q_{CA} &= \{p_{vi} : v_i \in Q_{FA}\} \\
\Sigma_{CA} &= \Sigma_{FA} \\
R_{CA} &= \{q_i : q_i \in F_{CA}\} \\
F_{CA} &= \{\} \\
M_{CA} &= \{\}
\end{align*}
\]

\( \rightarrow_{CA}^2 = \{p_{vi} \xrightarrow{x} r_{vi} \otimes p_{vj} : v_i \xrightarrow{x} v_j \in \rightarrow_{FA} \land p_{q_i} \in Q_{\Pi_{CA}} \land q_x \in F_{\Pi_{CA}} \}
\]

\[\cup \{p_{vk} \xrightarrow{\lambda} \varepsilon : p_{vk} \in F_{FA}\}\]

where \( \otimes = \begin{cases} . & \text{if } x \text{ is implemented by } C_A \text{ as synchronous interface} \\ || & \text{if } x \text{ is implemented by } C_A \text{ as asynchronous interface} \end{cases} \)

By this definition, for each transition rule \((v_i, x) \xrightarrow{v_j} \in \rightarrow_{FA}\) of the protocol \( P_A \) of the aggregated component \( C_A \) a new transition rule \( p_{vi} \xrightarrow{x} r_{vi} \otimes p_{vj} \) of \( C_A^2 \) is created. An example for this construction is shown in Figure 9.

Lemma 1 Let be \( \Pi_S^A = \phi_{P_A}(\Pi_S) \), where \( \Pi_S = C_A^2 \oplus \Pi_{CA} \). Then \( L(\Pi_S^A) = L(\Pi_S) \), where \( P_A \) is the protocol of \( C_A \).

Proof 1 (Sketch) By the architectural assumption in this section, \( C_A \) cannot (neither directly nor indirectly) call to its provided interfaces, because \( C_A \) has no required interfaces. Thus for any provided interface \( q_x \in F_{CA} \) of \( C_A \), it holds \( p_{q_x} \xrightarrow{\lambda} \varepsilon_{\Pi_S^A} \varepsilon \) where \( p_{q_x} \) is the initial state of interface \( x \) (otherwise a call to \( x \) would be always non-terminating). We prove now the following claim by induction sufficient for \( L(\Pi_S^A) = L(P_A) \):

For each \( w \in \Sigma_{P_A}^2 \), it holds \( v_0 \xrightarrow{w_{P_A}} v_k \) iff \( v_0 \xrightarrow{w_{\Pi_S^A}} p_{v_k} \).

\underline{if}: We only consider the case where the interface is implemented synchronously. The asynchronous
Then the derivation of case can be proven analogously.

**Case w = λ:** Then \( v_0 \xrightarrow{\lambda} p_a v_1 \) and \( p_{v_0} \xrightarrow{\lambda} p_{v_1} \) proves the claim.

**Case w = w' a** for a \( w' \in \Sigma_{\mathcal{P}_A} ^\star \), \( a \in \Sigma_{\mathcal{P}_A} \): Then \( v_0 \xrightarrow{w'} \mathcal{P}_A \ v_{k-1} \xrightarrow{a} \mathcal{P}_A \ v_k \). Thus \( v_{k-1} \xrightarrow{a} v_k \in \rightarrow_{\mathcal{P}_A} \) and therefore \( p_{v_{k-1}} \xrightarrow{a} p_{v_{k-1}} p_{v_k} \in \rightarrow_{\mathcal{P}_A} \) (by Definition 6).

Thus we have \( p_{v_{k-1}} \xrightarrow{a} p_{q_x} p_{v_x} \xrightarrow{\lambda} p_{v_k} \) (by the above observation)

\[
\text{= } p_k \quad \text{(since } \varepsilon \text{ is the left identify of .)}
\]

Now we show that a derivation in \( \Pi_{\mathcal{S}^A} \hat{=} \varphi_{\mathcal{P}_d}(\Pi_{\mathcal{S}^A^{Im}}) \) (where \( \Pi_{\mathcal{S}^A^{Im}} \hat{=} \Pi_{C_{\alpha}^A} \oplus \Pi_{C_{\beta}^A} \)) corresponds to a derivation of \( C_{\alpha}^A \oplus C_{\beta}^A \). In particular, our goal is to show that \( L(\varphi_{C_{\alpha}^A}(\Pi_{C_{\alpha}^A} \oplus \Pi_{C_{\beta}^A})) \subseteq L(\varphi_{C_{\beta}^A}(C_{\alpha}^A \oplus C_{\beta}^A)) \) for all internal components \( C_{\alpha} \) of \( C_{\beta} \), if the protocol of \( C_{\beta} \) has been checked. Such a result implies that replacing \( C_{\alpha}^A \) by \( C_{\alpha}^{Im} \) does not lead to new protocol violations except possibly for \( C_{\beta} \). In order to show a correspondence between derivation \( \Pi_{\mathcal{S}^A^{Im}} \) and \( \Pi_{\mathcal{S}^A} \), we need a correspondence between process-algebraic expressions \( t \in \mathcal{P}E X(Q_{\Pi_{\mathcal{S}^A}^{A}}) \) and \( t' \in \mathcal{P}E X(Q_{\Pi_{\mathcal{S}^A}^{A}}) \). Note that \( Q_{C_{\beta}^{C_{\alpha}}} \subseteq Q_{\mathcal{S}^{C_{\alpha}}} \cap Q_{\Pi_{\mathcal{S}^A}^{A}} \) because these states stem from \( \Pi_{C_{\alpha}} \). The main idea for defining the correspondence basically requires that sequences initiated by \( \Pi_{C_{\alpha}^{Im}} \) are sequences initiated by \( \Pi_{C_{\alpha}^A} \).

Intuitively, a \( t \) corresponds to \( t' \) iff forgetting the states not in \( Q_{C_{\beta}} \) leads to the same process-algebraic expression, and the same sequence of interactions to \( C_{\beta} \) can be initiated from \( t \) and \( t' \).

**Lemma 2** If \( L(\varphi_{\mathcal{P}_d}(\Pi_{\mathcal{S}^{Im}})) \subseteq L(\mathcal{P}_A) \), and there is a \( w \in \Sigma_{C_{\alpha}^A} \) and \( t \in \mathcal{P}E X(Q_{\Pi_{\mathcal{S}^{Im}}^{A}}) \) such that \( I \xrightarrow{w} \Pi_{\mathcal{S}^{Im}} \), then there is a \( t' \in \mathcal{P}E X(Q_{\Pi_{\mathcal{S}^{A}}^{A}}) \) with \( t \) corresponds to \( t' \) and \( p_{v_0} \xrightarrow{w} p_{v_1} \).

**Proof 2 (Sketch)** We use induction to prove the claim. If a word \( w \), with \(|w| = 1\) is created, a transition rule in \( C_{\alpha}^{Im} \) has to be used which has the form \( p_1 \xrightarrow{x} p_2 \otimes p_3 \), \( x \in L(\mathcal{P}_A) \), \( \otimes \in \{+,\|\} \).

Because \( C_{\alpha}^{Im} \) obeys the protocol \( \mathcal{P}_A \), there exists a transition rule \( p_{v_0} \xrightarrow{x} p_{v_2} \otimes p_{v_1} \) in \( \Pi_{\mathcal{S}^{A}}^{A} \) (more precise in \( C_{\alpha}^{\mathcal{P}_A} \)). Thus the term \( t' \) is constructed using this rule. \( t \) corresponds to \( t' \).

Assuming that the claim is true for words of length \( n \). A word \( w' = w \cdot a \) with \(|w'| = n + 1\), can only be constructed using a rule \( \delta \xrightarrow{a} p_3 \xrightarrow{a} p_4 \) in \( C_{\alpha}^{Im} \). A similar rule \( \delta' \) have to be part of \( C_{\alpha}^{\mathcal{P}_A} \), such that the same word \( w \cdot a \) can be constructed. While constructing \( w \) in \( C_{\alpha}^{Im} \) a term \( t \) have to be computed, encoding a situation where \( \delta \) is applicable. Because \( t' \) encodes the same situation, \( t \) corresponds to \( t' \).

**Corollary 1** \( L(\varphi_{C_{\beta}^A}(\Pi_{\mathcal{S}^{Im}})) \subseteq L(\Pi_{\mathcal{S}^{A}}^{A}) \).

Hence, a verification driver \( C_{\alpha}^{\mathcal{P}_A} \) can generate all sequences which are permitted by the protocol \( \mathcal{P}_A \) of \( C_{\beta} \).
Now it is possible to model check the scenario $a$ described in Section 4.

We generate a system abstraction $\Pi_S$ while unifying the behavior of the aggregated component $C_A$ and the verification dummy $C_M^A$ as described in Section 3 (an example is shown in Figure 9b). The resulting $\Pi_S$ is used to generate a Combined Abstraction for each protocol $P_i$ of each component $C_i$, where $C_i$ is included in the aggregated component $C_A$.

Now we have to prove that no protocol violation is missed while considering the verification driver $C_W^A$.

**Theorem 1** If $L(\varphi_{P_i}(\Pi_A^3)) \subseteq L(P_i)$ for each protocol $P_i$ of a component $C_i$ in $C_A$ (i.e., we have protocol conformance proven using the verification driver), and $L(\varphi_{P_A}(\Pi_{S_{Im}})) \subseteq L(P_A)$ (i.e., the component $C_{Im}$ uses $C_A$ according to the protocol $P_A$) then for each component $C_i$ of $C_A$, it holds $L(\varphi_{P_i}(\Pi_{S_{Im}})) \subseteq L(P_i)$ (i.e., the protocol is satisfied if $C_{Im}$ is used instead of the verification driver $C_W^A$).

**Remark 3** It is clear that if $L(\Pi_1) \subseteq L(\Pi_2)$ then $L(\varphi_{C_i}(\Pi_1)) \subseteq L(\varphi_{C_i}(\Pi_2))$.

**Proof 3 (Theorem 1)**

\[
L(\varphi_{P_i}(\Pi_{S_{Im}})) \subseteq L(\varphi_{P_i}(\Pi_A^3)) \text{ by Corollary 1 and Remark 3}
\]

\[
\subseteq L(P_A) \text{ by assumptions of Theorem 1 }
\]

Hence, all possible protocol violations by any implementation (without callbacks) can be found.

### 5.2 Components with unbound provided and unbound required interfaces

In this section we consider the scenario $b$ (cf. Section 4) of Figure 7b (allowing no callbacks).

For model checking a complete PRS is needed. Thus a verification driver $C_W^A$ and the abstraction of the aggregated component $\Pi_{C_A}$ are not sufficient, because there are still unbound required interfaces of $C_A$. We therefore add a SPRS $C_M^A$, called verification dummy. It is bound to the required interfaces of $C_A^R$.

The verification dummy $C_M^A$ has to accept all calls from the aggregated component $C_A$ to its unbound required interfaces $r_i \in R_{C_A}$, but imitate no callbacks to a $q_j \in P_{C_A}$. Thus $C_M^A$ imitates all possible base components, which could be used to complete the application using $C_A$. Here we define the verification dummy formally as SPRS $\Pi_{C_M^A}$. 

Thus the verification dummy $C_A^{\mathfrak{d}}$ of $C_A$ can be bound at $C_A$ and capture all behavior which is possible by a component $C_2$ implementing the context. Thus $C_A^{\mathfrak{d}} \oplus C_A \oplus C_2^{\mathfrak{d}}$ is a PRS.

**Theorem 2** An implementation $C_{\text{Im}}$ of the verification dummy $C_A^{\mathfrak{d}}$ can not initiate a protocol violation.

**Proof 4** Callbacks are forbidden. Thus no call to an interface of $C_{\text{Im}}$ can result in a call to $C_A$.

No protocol violation can appear.

**Corollary 2** Let be $\Pi_{S\text{Im}} \doteq C_1 \oplus \Pi_{C_A} \oplus C_2$, where $C_1$ is an implementation of $C_A^{\mathfrak{d}}$ and $C_2$ is an implementation of $C_A^{\mathfrak{d}}$. Violations of $P_i$ of $C_i$ can only be initiated by $C_1$ (captured by Theorem 1).
An example is shown in Figure 10. As we can see no protocol violation happens while verifying the protocols $P_{C_2}$, because all computable interaction sequences $w$ are contained in $L(P_{C_2})$.

6 Verification Context (Allowing Callbacks)

For humans it is usually difficult to discover protocol violations if callbacks are present, in particular if these are recursive. Thus it is an important task to find or exclude protocol violations for this case. For this reason we generate a general context, which contains all possible callbacks, too. As before we use a SPRS $\Pi_{C_A}$ representation to describe the behavior of the verification context.

Definition 8 verification context $\Pi_{C_A}$

A verification context is a SPRS $\Pi_{C_A} = (Q_{C_A}, \Sigma_{C_A}, \rightarrow_{C_A}, R_{C_A}, P_{C_A}, M_{C_A})$ where

- $Q_{C_A} = Q_{C_A} \cup Q_{C_A}$,
- $\Sigma_{C_A} = \Sigma_{C_A} \cup \Sigma_{C_A} = \Sigma_{C_A}$,
- $\rightarrow_{C_A} = \rightarrow_{C_A} \cup \rightarrow_{C_A} \cup \rightarrow_{\circ}$,
- $R_{C_A} = R_{C_A} \cup R_{C_A} = R_{C_A}$,
- $P_{C_A} = P_{C_A} \cup P_{C_A} = P_{C_A}$,
- $M_{C_A} = M_{C_A} \cup M_{C_A} = M_{C_A}$

To capture the semantics of callbacks we introduce the special set of transition rules $\rightarrow_{\circ} = \{p_q, \rightarrow p_v : p_q, p_v \in Q_{C_A} \land p_v \in Q_{C_A}\}$ (callback rules). The transition rules $\rightarrow_{C_A}$ allow all possible interactions with $C_A$. Using rules from $\rightarrow_{C_A}$ it is possible to generate any possible interaction $a \in P_A$ to a provided interface $q_i \in P_{C_A}$ of $\Pi_{C_A}$ (cf. Section 5.1). Using rules from $\rightarrow_{C_A}$ any external interaction of $C_A$ with the context is captured (cf. Section 5.2).

The semantics of callbacks are that any call to a required interface $r_i \in R_{C_A}$ of $C_A$ can result in a sequence of calls to any provided interface $q_j \in P_{C_A}$ of $C_A$. To represent this behavior we generate the transition rules $\rightarrow_{\circ}$. Informally this ensures that after each call to a required interfaces $r_i$ (mapped by $M_{C_A}$ to $p_r$) any of the provided interfaces $q_x \in P_{C_A}$ of $C_A$ can be called, too, using $p_v, x \rightarrow r_x \otimes p_k$ (contained in $\rightarrow_{C_A}$).

An example is shown in Figure 11. There a protocol violation is initiated if a call to $a$ could result in some way to a call of $b$ of component $C_2$. This counterexample (Figure 11c) precisely describes the problem observed in Figure 1. Thus the problem discussed in the motivation will be

---

7 For technical details on the construction of the counterexample we refer to [BZ08b, BZ08a].
The verification context captures the behavior of every possible implementation.

**Theorem 3** Let $C_{im}$ be an implementation of the verification context $C_A^o$ of the aggregated component $C_A$ and $P_A$ be the protocol of $C_A$. We consider $\Pi_{SIm} \doteq \Pi_{Cim} \oplus \Pi_{CA} \oplus \Pi_{Cim}$ and $\Pi_{SIm}^c = \varphi_{Cim}(\Pi_{SIm})$ (thus it contains only actions of $C_{im}$). If $L(\Pi_{SIm}^c) \subseteq L(P_A)$ and there is no protocol violation in $C_A$ (i.e., $L(\Pi_{SIm}) \subseteq L(P_i)$) for each internal component $C_i$ of $C_A$, then for each internal component $C_i$ of $C_A$, there is no protocol violation if $C_A$ is replaced by $C_{im}$ (i.e., $L(\Pi_{SIm}^c) \subseteq L(P_i)$).

**Remark 4** If no callback exists within the implementation $C_{im}$ of the verification context $C_A^o$ we can derive from Theorem 1 that all protocol violations are already described with the transition rules of $C_A^Q$.

In order to prove Theorem 3 the following lemma is required.

**Lemma 3** Let be $\Pi_{SIm} \doteq \varphi_{C_i}(\Pi_{SIm}^c)$ and $\Pi_S \doteq \varphi_{C_i}(\Pi_S)$, where $\Pi_{SIm}^c \doteq \Pi_{Cim} \oplus \Pi_{CA} \oplus \Pi_{Cim}$ and $\Pi_S \doteq \Pi_{Cim} \oplus \Pi_{CA} \oplus \Pi_{Cim}^o$. If $p_x \xrightarrow{w} p_{e_1}$ then $p_x \xrightarrow{w} p_{e_1}$, where $w \in \Sigma_{P_i}, t \in PE(\Pi_{SIm})$, $q \in$
Proof 5 (Lemma 3) Induction over the length \( n \) of the sequence of interactions \( w \).

1. \( n = 1 \): If \( p_{rs} \xrightarrow{w} \Pi_{S_{lm}}^C t_E \), with \( w \in \Sigma_p \), then a transition rule \( p_1 \xrightarrow{a} p_2 \otimes p_3 \) has to be used. This rule is part of \( C_A \), thus \( w \) can be generated in \( \Pi_{S_{lm}}^C \), too.

2. \( n = 2 \): If \( p_{rs} \xrightarrow{a} \Pi_{S_{lm}}^C t \xrightarrow{b} \Pi_{S_{lm}}^C t_E \), with \( a, b \in \Sigma_P \). It can be refined to \( p_{rs} \xrightarrow{a} \Pi_{S_{lm}}^C t \xrightarrow{\lambda} \Pi_{S_{lm}}^C \) \( t' \xrightarrow{b} \Pi_{S_{lm}}^C t_E \). A transition rule \( p_1 \xrightarrow{a} p_2 \otimes p_3 \) and a transition rule \( p_4 \xrightarrow{b} p_5 \otimes p_6 \) have to be used to generate the counterexample, both rules are contained in \( -C_A \), thus they are also available in \( \Pi_{S}^C \) (\( \otimes, \otimes' \in \{., ||\} \)). \( t \xrightarrow{\lambda} t' \) can be computed within \( \Pi_{S_{lm}}^C \) only in three cases:

- To compute \( t \xrightarrow{\lambda} t' \) only rules of \( -C_A \) are applied. Thus the derivation is also computable in \( \Pi_{S} \) (especially \( t = t' \)).

- To compute \( t \xrightarrow{\lambda} t' \) rules of the verification context \( C_A^c \) are used but no callbacks. By Remark 3 the derivation is also computable in \( \Pi_{S} \).

- To compute \( t \xrightarrow{\lambda} t' \) a callback can be used. Thus the derivation can be refined to \( t \xrightarrow{\lambda} t_{r_j} \xrightarrow{\lambda} t_{q_k} \xrightarrow{\lambda} t' \), where \( t_{r_j}, t_{q_k} \in PEX(Q_{S_{lm}}^C) \) and in \( t_{r_j} \) a transition rule is applicable to a required interface \( r_j \) of \( C_A \) and \( t_{q_k} \) a transition rule is applicable to a provided interface \( q_k \) of \( C_A \). Because of the rule \( p_{r_j} \xrightarrow{\lambda} p_{q_k} \in -C_A \) this derivation is also possible in \( \Pi_{S} \).

3. \( n = m \): The lemma is valid for any interaction sequence: \( p_{rs} \xrightarrow{w} \Pi_{S_{lm}}^C t \).

4. \( n = m + 1 \): If \( p_{rs} \xrightarrow{w} \Pi_{S_{lm}}^C t \xrightarrow{a} \Pi_{S_{lm}}^C \varepsilon \), then a rule \( p_1 \xrightarrow{a} p_2 \otimes p_3 \) has to be applied. This rule is part of \( -C_A \), hence this derivation is also computable in \( \Pi_{S_{lm}}^C \), because the first part is covered by the induction hypotheses.

Corollary 3 \( L(\varphi_{C_A}(\Pi_{S_{lm}}^C)) \subseteq L(\Pi_{S_{lm}}^C) \)

Proof 6 (Theorem 3) Analogous to Theorem 1.

7 Related Work

Test techniques like unit testing are compositional, they can find errors, but do not prove their absence. Thus they are only capable of validating the behavior of a single (aggregated) component and not to check whether the component may work in an unknown application. The definition of pre- and postconditions has the problem, that it is only rarely possible to determine (statically) whether a condition can never be violated, if the context is unknown.
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 [Nie95].

In [ZS06] recursion is modeled by CFG, so only sequential behavior is considered. Moreover recursive callbacks are already considered for complete systems. [Obd02] verifies restrictions on call graphs in Java programs, while using CFG it is possible to formulate the restrictions in LTL. In [AM04] pushdown languages are used to represent program behavior, these can be used to model the protocol conformance, too.

Schmidt et al. [HBPR02] propose an approach for protocol checking of concurrent component-based systems. Their approach is also FSM-based and unable to deal with recursive callbacks. In [PP08] behavioral protocol conformance for complete systems is used to describe a problem similar to ours.

To check compatibility of components another approach is suggested by [ME04]. There, the input and output languages of components are considered, which is a stronger requirement than our compatibility constraint (interaction protocols).

Some of these works can handle compositional contracts. In contrast to them we use in our work the composition to aggregate not only the behavior but allow to reduce the contracts which have to be checked in the next compositional step. Moreover, our approach is more powerful than the mentioned approaches, because we use a representation which can capture unbound recursion and unbound parallelism [BZ08a]. This is important since (recursive) callbacks or reference calls in the source code cannot be excluded [ZS06]. Moreover our approach does not reduce the model checking problem to a finite state model checking [BZ09]. Hence we are enabled to find protocol violations in general.

Besides interaction protocols there are other works considering aggregated components. In [Gia95] the hierarchically composed components are summarized while considering only selected actions. Simplifications of subsystems are performed by minimising the representation with respect to observational equivalence. Callbacks are not allowed. In [CCG+04] labelled transition systems are used to represent the (wished) behavior of (procedural) components written in C. The developed model checker MAGIC can use these abstractions in a compositional way to consider component substitutability. In the component verification approach [CvH99] aggregated components are considered explicitly. In contrast to our approach, the constraints of the included components can not be eliminated. Thus the number of constraints of the aggregated components
may grow in the size of the aggregated component.

To our knowledge no work considers composition while allowing both callbacks (needed for e.g., SOA) and the reduction of interaction constraints of subcomponents when the constraint of the aggregated component can always be proven as always fulfilled.

8 Summary

In this paper we have shown how to support the development process of reliable components and applications, respectively. We use interaction protocols as constraints for the usage of components and define a formal approach based on model checking. This allows the system architects or component developer to check the implementations, even if the components are implemented in different programming languages. They can check whether the implementation of every possible application matches the constraints of the protocols of the components which are aggregated. In contrast to testing procedures we can prove absence of errors, because we find every possible error. This is an important new option.

Moreover the development might speed up, because if the user of a component has to interact conformly with the protocol, the developer of this component does not have to implement checks and catch for all errors which can be triggered from any unexpected interaction.

The approach allows to check the protocols of components aggregated to a larger component. This common step during the development is now suitably supported by our verification process. Thus, a permanent verification of each composition or development step is possible. The result is a reliable aggregated component which can be composed easier. As a side effect the costs of the verification can be reduced because interaction protocols need not to be checked again, if they are integrated into an aggregated component.

Moreover the constructed counterexamples of an aggregated component can be used to evaluate the design of the application in an early phase as well as they can point the developers of the currently unfinished components to possible implementation errors. Both will lead to a more predictable implementation of the application, where errors could be discovered earlier.

We combined here the assumption about the application software design with a verification approach. Hence our verification approach can be applied to consider B2B-scenarios, client server and tier architectures, Service-oriented Architectures and many more.

First observations of an industrial case study (at the OR Soft Jänicke GmbH⁹) show that our

⁹http://www.orsoft.net
approach leads to the expected improvement of model checking speed and a better quality of the results (because of their local character).

In future works we will focus on the generation and validation of protocols. It could be interesting to evaluate, which protocols can be defined automatically for an aggregated component, because then the developers are disburdened from this task. Thus the developer of a component only has to define a protocol for a single component. It will be generated automatically for each aggregated component.

If we think ahead to support the development even more we should integrate the verification process in the modelling of the application. Furthermore it seems to be possible to derive protocols semi-automatically from the pieces of information available in the design phase. Then errors can be found even sooner.

We are grateful to OR Soft Jänicke GmbH for providing us with industrial case studies.

References


[PP08] Pavel Parizek and Frantisek Plasil. Modeling of component environment in presence of callbacks and autonomous activities. In Richard F. Paige and Bertrand Meyer, editors,


Information about the Author(s)

Andreas Both
Institut für Informatik
Universität Halle
06099 Halle, Germany
E-Mail: andreas.both@informatik.uni-halle.de
WWW: http://swt.informatik.uni-halle.de/both

Wolf Zimmermann
Institut für Informatik
Universität Halle
06099 Halle, Germany
E-Mail: wolf.zimmermann@informatik.uni-halle.de
WWW: http://www.informatik.uni-halle.de/~zimmer