Abstract—In several works a method was suggested to overcome the lack of signature-based composition currently enabled in component-based and service-oriented architectures (SOA). Several approaches allow to encode non-functional properties of a single component in a contract (component protocol) where the remote calls to a component are taken into consideration. Component protocols ensure that bugs or unsafe behavior caused interaction sequences are obeyed. Encoding business rules works fine as these contracts can be derived from human knowledge only and have to be defined manually, too.

In this work we will show, how such unsafe behavior within source code can be discovered and prevented by automatic component protocol generation and model checking techniques.

Keywords-component-based software engineering, protocol generation, protocol conformance checking, verification, model checking, component composition

I. Motivation

In current software development processes a growing set of components is reused. Often components are composed for new software. Currently, component frameworks (like Microsoft .NET, Sun Java EE and web services) provide a signature-based composition only. Thus, non-functional properties are not considered. Hence, this composition method is error-prone. Several problems can appear while using stateful or stateless components, for example, a component should be used in a specific way to obey a business rule (e.g., add products to basket, confirm order, pass credit card data). To overcome these problems several approaches are suggested. An easy to use approach contains contracts for each component to publish the allowed interaction sequences. These usage contracts are called component protocols [3], [2].

These component protocols are often defined by: (1) the component developer (if a problem is known [3]), (2) the component designer (if a start for development should be defined), or (3) a business analyst (to establish a specific workflow that should be obeyed [4]). These defined protocols can be proven automatically using model checking leading to a program execution trace (or run) where the demanded behavior is not fulfilled [4]. However, while using only component protocols that are defined by humans, bugs in components are not considered in general. Thus, the composition of components is only reliable in the sense of the properties a responsible human knows or recognizes are obeyed. In previous work it is assumed that all components are reliable if the corresponding component protocols are obeyed. This is the same situation that appears in the design-by-contract approach [13] : If not all properties are captured, the composition and execution might not be reliable.

It is clear that not all problems that can ever appear (Turing completeness) can be avoided. Nevertheless, the goal should be to avoid as many of them as possible. In particular, this is needed if the considered erroneous component is a legacy component. Then a correction of a component source code might be difficult or impossible (e.g., glassbox component).

In this paper we will show how model checking techniques can be used to discover unwanted situations in a single component (written in Java) without knowing the future context of the component. For this purpose our tool Halle’s exact Value Range Extractor (HalVRE, [21]) is used. The process is divided into the following steps (Figure 1).

1. Define unsafe situations and their patterns (e.g., null-pointer-dereference of attributes or assertion failures).
2. Translate Java source code using jMoped [26] to reachability models\footnote{This translation is not restricted to Java. Also other languages can be handled. Some programming languages like C for embedded systems can be directly expressed using PDS [22]. In such a case no translation is necessary.} (symbolic pushdown system (SPDS)) considering the unsafe situations.
3. Solve the reachability model using HalVRE and compute an automaton describing all unsafe situations.
4. Construct the general component protocol automaton by considering all unsafe interaction sequences.
5. Automatically verify the given component protocols against the usage in the composition of components.

Step 5 encapsulates a verification process for protocol conformance checking defined in earlier work [3]. It does not need the source code and is capable of dealing with unbounded parallelism and unbounded recursion while using a powerful representation of the abstract behavior. Thus, a rugged composition can be ensured if sound component protocols exist.

The main contribution of this paper is the automatic computation of component protocols (steps 2–4). For this purpose specific properties are considered while using model
checking techniques to evaluate the source code of a single (Java) component. The results are put in correlation to avoid forbidden situations and interaction sequences. This method leads to a new component protocol preventing all corresponding interaction sequences.

The paper is organized as follows. In Section II the foundations are described. This contains the definition of the contracts (component protocols) used in this paper and HalVRE. In Section III our approach is presented. It is divided in two phases. First, pieces of information about the possible situations are computed automatically. Second, these situations are evaluated automatically and a contract is generated. The paper finishes with a consideration of the related work in Section IV and the conclusion and future work in Section V.

II. FOUNDATIONS

A. Components/Web Services

The components (e.g., web services) in this work are represented by using a model similar to the UML component model. They have to use required interfaces and implement provided interfaces. The latter are usually predefined in an interface description (e.g., WSDL). Our component model is shown in Figure 2. Components are singletons in this paper. This restriction is for simplification reasons only, references (instances) of components can be handled too [28].

Here we assume that it is known whether a service call should be implemented blocking (synchronous, short: sync), or non-blocking (asynchronous, short: async). In [2] it is described how method calls are handled if it is not known, how they are implemented. We denote this information within the interface description (cf. \( \text{invKind}_i \) in Fig. 2).

We also denote the invocation implementation at the interface definitions and we accept implementations in any imperative or object-oriented programming language [3] (restricted to Java in this paper). While calling a method, we assume that the global data that will be accessed during the method execution is locked using semaphores. Thus, only one method is able to access a global variable at a point in time. This restriction will be lifted in future work.

B. Component Protocols

A component protocol (short: protocol) describes the allowed use of all callable operations (cf. interfaces) of a component. To obey the character of many component systems, a protocol is defined using the provided interface operations only. It might be used to obey a business rule (BPM) that is implemented by a component (e.g., first login, then add items to the basket, finally add the credit card information, [4]) or to prevent an unsafe situation (like a division by zero appearing because of control-flow dependencies [3]). In general, a protocol describes a (usage) workflow of the considered component that matches the intention of the developer. For this reason it is often called usage protocol. It can be used to dynamically verify (incoming remote) invocations, and also to statically verify, if the 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. In this work, we only consider static component behavior verification, as it is defined in [2]. Thus, the purpose of component protocols is to check the behavior before a component is executed. Thereby, exceptions appearing during actual user interactions are prevented. For this purpose we verify the component software at the design, composition, or deploy time, however not at the execution time.

We use finite state machines (FSM) to formalize component protocols\(^2\). The FSM \( \text{FSM} = (Q_P, \Sigma_P, T_P, I_P, F_P) \) is defined as usual, i.e., \( Q_P \) is a finite set of states, \( \Sigma_P \) is a finite set of atomic actions, \( T_P \subseteq Q_P \times \Sigma_P \times Q_P \) 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.

\(^2\)Using more powerful protocol representations is uncommon for describing the interface usage. Note, the FSM is only the contract of a component. It does not restrict the infinite behavior represented by the abstract behavior (Section II-C).
and $F_P \subseteq Q_P$ is the set of final states. $P$ defines a regular language $L(P)$. Note that $\Sigma_P$ contains all provided methods published via the component interface descriptions, e.g., $\Sigma_{C_2} = \{a, b, c, d\}$ of component $C_2$ in Fig. 3. Hence, a component protocol is only based on published information. It obeys the properties of component systems that are used in industrial environments and provides more flexibility while allowing the exchange of the implementation of a component, as the behavior of the component is not included. Moreover, it is possible to define more than one component protocol. These are main distinctions to other works.

Remark 1: E.g., in [15] the protocol is a specification of a component behavior. Hence, the abstract behavior (outgoing interactions) is contained in the (behavioral) protocol, too. Therefore, this kind of a protocol is a specification of the actual behavior. However, a correlation to the actual implementation of the component is not checked. In contrast, the component protocols used here specify a contract of allowed usage interactions of a component only.

A possible protocol of the component $C_1$ is shown in Figure 4 in a graphic representation.

C. Abstract Behavior and Protocol Conformance Checking

To check the protocol conformance of the considered software we also need the abstract behavior of each component. We use a representation – named Process Rewrite Systems (PRS) [12] – which is capable of representing unbounded recursion and unbounded parallelism. In [3], [2] an automatic verification process is defined, where the source code is translated into a conservative abstract behavior $\Pi$ (PRS) that is not Turing-powerful. Thus, it can be checked $L(\Pi) \subseteq L(P)$ [3], where $P$ is the considered protocol. This (conservative) protocol conformance checking results in a set of counterexamples. These describe (symbolically) the interaction sequences leading to forbidden situations. If no counterexample was found, the software is error-free in the sense of the given component protocols.

D. Tools and Definitions for SPDS

$\mathfrak{P} = (S, \Gamma, \rightarrow)$ is called pushdown system iff $S$ is a set of states, $\Gamma$ is a finite set of symbols (pushdown alphabet) and $\rightarrow \subseteq (S \times \Gamma) \times (S \times \Gamma^*)$ is a set of transitions. A large subset of modern programming languages, resp. the complete ISO-C semantics can be expressed using PDS [22].

For symbolic pushdown systems (SPDS) the transitions are described symbolically using relations [25]. In this way it is easier to specify a PDS. The model checker Moped [25] can check the reachability property of a SPDS that is described using the model description language Remopla [11]. Remopla is similar to an imperative programming language and can be derived automatically from Java source code using jMoped [26]. As well as in Java or C, in Remopla primitive data types are restricted to finite domains. In Listing 1 you can see a sample Java source code which was translated to Remopla (Listing 2) using jMoped.

Listing 1: Segment of Java code for a component $C_2$.

```java
class C2 {
    static Object x, y;
    void main(String s[]) {
        int q = undef; c(q);
        void c(int p) {
            if (p>1) x = null;
        }
    }
}
```

Listing 2: Segment of generated SPDS of Listing 1.

```java
int C2_x(16);                   # pointer into the heap
int C2_y(16);
module void C2_main(int v0(16), int v1(16)) {
    int s0(16);
    C2_main_V7: s0=undef;
    C2_main_V8: C2_c_1_V(s0);
    C2_main_V9: return;
}
module void C2_c_1_V(int v0(16)) {
    int s0(16);
    s0=v0, s1=s0;
    C2_c_1_V1: s0=1, s1=s0;
    C2_c_1_V2: if (s1<s0) goto C2_c_1_V9;
    else -> skip; fi;
    C2_c_1_V5: s0=0, s1=s0;
    C2_c_1_V6: C2_x=s0, s0=s1;
    C2_c_1_V9: return;
}
```

Let $\text{Vars}_q := \text{globals} \cup \text{loc}_q$, $\text{EXPR}_{\text{Vars}_q}$ be the arithmetic expressions over these variables and $f^q : \text{Vars}_q \rightarrow \mathbb{N}$ a variable valuation. And $[e_1]_{f^q}$ the expression evaluation of the expression $e_1 \in \text{EXPR}_{\text{Vars}_q}$ using the variable valuation $f^q$. The most important Remopla statements are:

- $x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n$; where $x_i \in \text{Vars}_q$ is a synchronous parallel configuration change, which assigns each variable $x_i$ the value of $[e_i]_{f^q}$.
- $p(e_1, e_2, \ldots, e_n); k$ is a call to module $p$.
- return $e_i$ is a termination of a module while returning the value $[e_i]_{f^q}$.
- goto L; is an unconditional jump to label L.
- if :: $e_1 \rightarrow s_1; \ldots; e_n \rightarrow s_n; \ldots; e_m \rightarrow s_m$; fi is a conditional statement where $[e_i]_{f^q} \in \{0, 1\}$. One random statement $s_k$ with $[e_k]_{f^q} = 1$ is executed.

Comments starts with # and ends at the end of the line. Currently jMoped only supports bounded integer sizes (as it is typical for programming languages) and no garbage collection. HalVRE can be used, to compute exact variable evaluations in Remopla. Further details to Remopla, Moped, HalVRE and jMoped can be found in [14], [8], [20].

III. Compute Component Protocols

In this section, we will show how component protocols can be generated automatically. This is done while using a modified version of the Moped model checker to determine unsafe situations. While putting the computed information...
into correlation, a component protocol can be generated automatically.

Besides other situations, a behavior is unsafe if an execution trace exists leading to an uncaught exception, an assertion failure, or a null-pointer exception. These execution traces are discoverable in Remopla (and therefore in Java source code) using Moped\(^4\). For the terms of simplification we consider here possible null-pointer exceptions to show the applicability of our approach. With few exceptions, null-pointer analysis currently is done by run-time checks, or statically by abstract interpretation or manual verification of provided annotations. Null-pointer exceptions might be raised while calling a reference variable that is accessible globally and might be set to null by another method. This subject was also chosen because jMoped already provides support for finding such situations. In Figure 3 a motivating example is shown. There the component \(C_2\) provides the operations \(a, b, c\) and \(d\). It might crash while calling operation \(c\) after operation \(b\). This is caused by the statement in line 11 in Figure 3 setting the global component variable \(x\) (providing access to another component) to \(null\). While \(x\) is used in \(b\), any interaction sequence has to be forbidden as a precaution, where \(b\) follows \(c\) (unsafe situation). On the other hand \(x\) and \(y\) should not be used (method \(b\)) before the pointer to the component is initialized (method \(a\), using function getRef). Globally accessible reference variables of a component \(C\) are summarized in the set \(\text{Ref}_C\). Thus, a possible safe protocol of component \(C_2\) (\(\text{Ref}_{C_2} = \{x, y\}\)) could be \(P_{C_2}\), cf. Figure 4. First, the initialization (\(a\)) has to be performed. Thereafter, \(b\) and \(d\) are allowed to call, while finally the reference variable \(x\) has to be set to \(null\) (\(c\)).

Discovering all these dependencies and possible problems, as well as the generation of the protocol by hand, will be complicated and error-prone. For example, it is not clear whether \(P_{C_2}\) is as general as possible. Therefore, we describe in the following an automatic approach based on the model checker Moped to compute component protocols (cf. Fig. 5).

A. Discover Unsafe Behavior

To identify possible null-pointer exceptions, we have to find out whether a reference is \(null\) or points to a valid object just before using the reference.

We need to answer three questions to classify the (un)safe usage of reference variables for each method:

1) Which global variables are initialized within the currently considered method?
2) Is a global variable changed in the considered method?
3) Did a situation exist, where a global variable is killed?

After computing these properties, it is possible to correlate the particular methods and compute a general safe protocol of the considered component.

1) Calculate Initializations: We have to distinguish two kinds of initializations. The first one is a setting of variable values for primitive data types in Java. In Listing 3 you can see this kind of initialization for global variables (heap) and the class variable \(x\) of component \(C_2\) from the Java sample of Listing 1.

JMoped generates automatically initialization code for Java variables in this way. Local variables are initialized in their first occurrence on the left-hand side of an assignment within a module. The second kind of initializations are the assignments of a created object to a reference. In Remopla references are integer values into the heap array. Thereby, the \(null\) pointer is represented by an integer with the value 0. In our model, components are not destructible. Thus, only reference destructions have to be captured. So far, we can use our previously developed tool HalVRE to find the points of initialization of reference variables. This method successfully discards such initializations that fails and would never successfully initialize a reference.

Let \(x_i\) be the reference variables of a component and \(f\) a procedure. Then we create for each \(f\) a separate Remopla model \(RM_f\) with random and uninitialized parameters as shown exemplary in Listing 5. This model represents all possible executions including all possible variable valuations. In Remopla an initial procedure has uninitialized parameters by default (cf. \(C2\_main\) in Listing 3), so it is sufficient to create automatically a new Java main method, which calls all the methods to ensure that these procedures are modeled by JMoped\(^5\) (step 1 in Figure 5). An example of such an automatic created adapter for component \(C_2\) (see Figure 3) is shown in Listing 4. Then JMoped is used to generate a complete Remopla model which includes all modeled Java methods (step 2 in Figure 5). After this we can easily extract a single module \(f\) from this complete \(RM\) model and set

\[\text{Listing 3: Automatically generated initialization code.}\]

\begin{verbatim}
1 module void jinit() {
2     ptr=1, forall i in (0..65535): heap[i]=0;
3     C2.x = 0; C2.y = 0;
4     goto C2_main; }
\end{verbatim}

\[\text{Figure 5: Automatic generation of } RM_f \text{ (step 2 of Fig. 1).}\]

\[\text{But only sequential behavior is considered in this case.}\]

\[\text{Otherwise JMoped will drop out syntactical unreachable procedures.}\]
Listing 4: Java adapter for \( RM_f \) generation of \( C_2 \) (Fig. 3).

```java
class C2 { static Object x,y;
    void main(String a[]) {
        int i=undef;
        a(i); b(); c(i); d(); }
    void a(int q) { ... }
}
```

Listing 5: \( RM_a \) of \( C_2 \) (Fig. 3) and its exact value ranges.

```java
int heap[65536];
int ptr[16]; # heap--pointer to the next free space
int C2_x[16]; int C2_y[16]; # pointer into the heap
init jinit; # initialization module
module void jinit() { ptr=1, # ptr=[1]
forall i in (0.65535): heap[i]=0; # heap[∗]=0
C2_x=0; # C2_x=[0]
C2_y=0; # C2_y=[0]
goto C2_a_L1_V; }
module void a(int v0[16]) { # v0=[0,65535]
int s0[16]; # s0=[0,65535]
C2_a_L1_V1: C2_x=s0; # C2_x=[23]
C2_a_L1_V2: s0=getRef_C3(); # s0=[28]
C2_a_L1_V3: C2_y=s0; # C2_y=[28]
C2_a_L1_V4: return: }
```

Table I: Analysis results (step 3) for component \( C_2 \).

<table>
<thead>
<tr>
<th>method</th>
<th>initRef</th>
<th>usage</th>
<th>canNull</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variable y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( M_R = \text{modules}(R) \) and labels \( L_m = \text{labels}(m) \). Further let be \( x \in R \) a chosen reference variable of \( R \), \( m \in M_R \) a chosen module (Remopla procedure), \( l \in L_m \) a chosen label, and \( E_l \) the set of occurring expressions an Label \( l \). Then we define usage definitions as:

\[
isused_x(m) = \text{true iff } \exists l \in L_m: \exists e \in E_l: x \in e
usage_x = \{ m | m \in M_R \land isused_x(m) \}
\]

For the above example it is calculated \( usage_x = \{ b \} \) and \( usage_y = \{ b, d \} \).

3) Calculate Object Reference Destructions: Now it will be checked which references may be destroyed (here, set to null). For this purpose we reuse the computed exact value ranges of \( RM_f \). If a reference variable can be null in \( RM_f \), then it might lead to an unsafe situation. Therefore, we add such methods \( f \) to the set \( canNull_x \). An initialization procedure \( a \) forms an exception, while it is assumed that no out-of-memory exception occurs during initializations. Such initialization procedures will not be taken as \( canNull_x \). The general approach for computing all \( RM_f \) of a component \( C \) is defined in Section III-A1.

The model \( RM_a \) of \( C_2 \) in Figure 3 is shown in Listing 4. Using \( HalVRE \) we find labels (and modules) where \( x \) can be null. Excluding initialization procedures \( initRef_x = \{ a \} \) results in \( canNull_x = \{ c \} \).

Remark 2: Although it is not clear if the branch of method \( c \) is actually used during the actual execution, \( x \) is marked as \( canNull_x \) for any execution of \( c \). In other words, while starting the execution of \( c \) it is already assumed that \( x \) will get the value null. This ensures that the concurrent execution of two or more methods will not result in an unexpected situation.

The complete results of the three calculations in this section considering the motivating example in Figure 3 is shown in Table I. Method \( a \) of component \( C_2 \) is the only one initializing the reference variables, while \( b \) uses the parameters for remote procedure calls. Only \( c \) may set \( x \) to null, and method \( d \) has nothing to do with the reference variables of the considered component. These separations were chosen for the purpose of simplicity in the example. In a general case it is also possible that all of these statements are contained in one method.

B. Generate Component Protocols

In this section, we will describe our approach to generating a component protocol from the properties computed in the previous section. It is clear that these situations have to be prevented where a reference variable is used before it

6Consider, that it is also possible to find reinitialisations using this method just by weakening the precondition that the variable \( x \) can have the value null \((0 \in \text{Range}(x))\) on label \( l \).
is initialized or after it is dereferenced (set to null). On the other hand, the component protocol has to be as general as possible to avoid restriction on the usage of components that would hamper the work of the developers.

To represent the behavior we describe here a two-phase process. In the first phase, we use a template protocol automaton (FSM) to compute for each reference variable \( x \) of the considered component \( C \) a separated partial protocol \( P_{C,x} \) obeying the restriction of \( x \). In the second phase all \( P_{C,x} \) are automatically conflated (intersection of protocols) to one general protocol \( P_C \), s.t. \( P_C \) prevents all unsafe situations. It is published together with the component interface description and used in the protocol conformance check during the composition composition.

1) Generate Protocol for one Variable: To represent the general protocol that prevents unsafe use of a reference parameter \( x \), we use the protocol \( P_{C,x} = (Q, \Sigma, T, I, F) \), where \( Q = \{v_0, v_1, v_2\} \), \( \Sigma = \{f : f \text{ is a provided method of the component } C\} \), \( I = v_0 \), and \( F = Q^f \). To define the set of transition rules \( T \) we use a shortcut to access the methods that neither influence nor use the current reference variable. We call this set \( \text{dontCare}_x \). It is defined by \( \text{dontCare}_x = \Sigma \backslash \text{initRef}_x \backslash \text{usage}_x \backslash \text{canNull}_x \).

The transition rules are defined by \( T = T_0 \cup T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5 \cup T_6 \), where:

\[ T_0 = \{(v_0 \rightarrow v_0 : f \in \text{canNull}_x \cup \text{dontCare}_x)\} \]

\[ T_1 = \{(v_0 \rightarrow v_1 : f \in \text{initRef}_x \backslash \text{usage}_x \cup \text{canNull}_x)\} \]

\[ T_2 = \{(v_1 \rightarrow v_1 : f \in \text{initRef}_x \cup \text{dontCare}_x)\} \]

\[ T_3 = \{(v_1 \rightarrow v_0 : f \in \text{canNull}_x)\} \]

\[ T_4 = \{(v_1 \rightarrow v_2 : f \in \text{usage}_x \backslash \text{canNull}_x)\} \]

\[ T_5 = \{(v_2 \rightarrow v_2 : f \in \text{initRef}_x \backslash \text{usage}_x \cup \text{dontCare}_x)\} \]

\[ T_6 = \{(v_2 \rightarrow v_0 : f \in \text{canNull}_x)\} \]

The calculation of the sets formalizes the intuition described before. It ensures that it never holds that method containing two or more operations on the same global reference variable. The resulting template protocol \( P_{C,x} \) is shown in Figure 6. We call this FSM \textit{template protocol automaton}.

Remark 3: In this paper only a single usage of a specific global variable within a specific method is considered as safe (others are excluded, cf. \( T_0 - T_6 \)). We choose this restriction with respect to the paper length. However, the consideration of such situations is possible using dataflow analyses.

7If programming languages are considered having no automatic garbage collection, then it might be valid \( F = \{v_0\} \).
protocols (the contracts) are finite state machines too, the
overall expressiveness of this verification process is higher.
These properties prepare the ground for the application in
industrial practice. However, there the finite-state protocols
(contract) have to be defined manually (usually top-down).

In [16] a behavioral protocol is computed bottom-up (with
several restrictions), i.e., an abstract behavior is computed.
However, this approach is very similar to our earlier work,
where abstracted behavior is computed automatically [3].

While using component protocols, discovered unsafe in-
teraction sequences (possible bugs) can be prevented au-
tomatically. To the best of our knowledge there exists no
other approach targeting this subject before the deployment
while using formal methods. In [18] contracts based on
state machines are generated by hand to predict component
reliability. The authors do not handle adequate parallelism
and recursion. Other approaches use test cases, e.g., to
ensure a requested behavior before the binding of a new
component [7]. Although these approaches are justifiable, a
component developer or user is not able to see the interaction
constraints. Moreover, formal methods prove the absence of
errors. Considering Java, for example, the tools jMoped/-
Moped [25], [26] and JavaPathFinder [6] are well known.
Other works use similar tools to evaluate the source of
(monolithic) software [17], the context and usage of a piece
of software must be known to discover faults. Moreover, here
it is possible to define more than one component protocol.
These are main distinctions to other works.

Remark 4: E.g., in [15] the protocol is a specification of a
component behavior. Hence, the abstract behavior (outgoing
interactions) is contained in the (behavioral) protocol, too.
Therefore, this kind of protocol is a specification of the
actual behavior. However, a correlation to the actual imple-
mentation of the component is not checked. In contrast, the
component protocols used here specify a contract of allowed
usage interactions of a component only.

Our earlier work assumes that a protocol is created by
a developer to prevent an error [3] or by a component
designer to encode desired workflows (e.g., business rules)
[4]. However, as shown in this paper, our current approach
does not clash with the earlier one, it extends them.

Here, we use Moped/jMoped as they provide several
significant advantages. In contrast to the model description
language Promela (the input language for the model checker
SPIN [9]), Remopla supports unbounded method calls and
recursion. However, it considers only synchronous config-
uration changes instead of parallel processes [26]. For the
purpose of model checking Java, the SPDS (in Remopla
syntax generated by jMoped) can be optimized using our
tool Symbolic Pushdown System Improver [24], [19], [23].
These techniques can also be applied here, but have to be
adapted in a straightforward way. Using such optimizations,
one can handle larger programs in less time.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new and original
approach targeting component protocols. In contrast to our
earlier and to other approaches the component protocols
(i.e., independent contracts) are generated automatically
(bottom-up), here. This is done using a model checker for
Java components checking for unsafe situations, even though
the approach is language independent. Thereafter, they can
put it into correlation and encoded into a new compo-
nent protocol. Nevertheless, the backward compatibility is
ensured, so combination with component protocols using
other processes is possible. The automatic approach provides
a significant benefit, while releasing developers from the
laborious and cumbersome work of thinking about possible
problems. Moreover, once defined template protocols can be
reused in any other component and thereafter will raise the
overall level of safety.

Our approach is leading to a prevention of unsafe situa-
tions especially while considering legacy components. In this
case the execution of the unadaptable source code is hedged.
However, even if the source code is adaptable an adaption
might be uneasy and too expensive. In these situations a
component protocol can ensure safe composition (e.g., in
a SOA) and provide a flexible and economical method.
Here, we have exemplary shown this while considering
null-pointer exceptions. A similar approach can be used to
consider the file life cycle (open, write, close) or database
connections, which are often used in industrial praxis. How-
ever, more complicated template protocols are possible, too.

Although it is possible to use tools other than HalVRE
(based on Moped) to determine the required information,
we prefer this tool as it is a model checker (calculating
stronger results than a test tool) and provides several options
for optimizations, extensions, and querying. This approach
enables one to consider more unsafe situations and forbid
them using protocols. This will increase the non-functional
property safety of the components considered with our method and lead to a more rugged composition in component software or service-oriented architecture. Note that using model checking techniques with push-down semantics helps to prevent false negatives (counting, recursion).

In future work, we will research more methods to discover (data dependent) unsafe situations. Thereby, a focus attach importance to the precision of the generated protocols. In this context, parallel behavior within a component will be considered too.

REFERENCES


