

Automatic Component Protocol Generation and Verification of Components

Andreas Both and Dirk Richter
Institute of Computer Science, University of Halle
06120 Halle (Saale), Germany
{andreas.both, dirk.richter}@informatik.uni-halle.de

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 ensures 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 models¹ (*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

¹This 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.

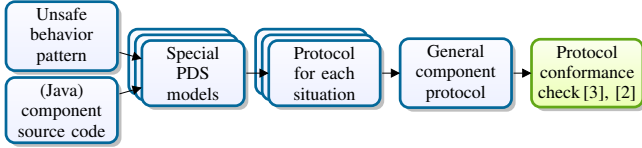


Figure 1: Concept of automatic component protocol generation and verification.

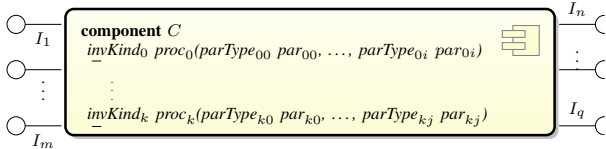


Figure 2: Component model.

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. $invKind_i$ 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

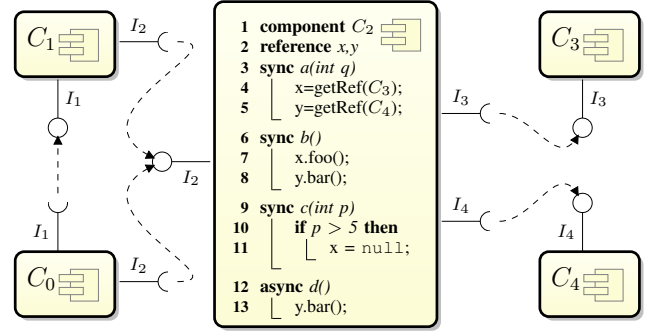


Figure 3: Component software consisting of five components. Only the source code of component C_2 is accessible.

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 protocols². The FSM $P \hat{=} (Q_P, \Sigma_P, T_P, I_P, F_P)$ is defined as usual, i.e., Q_P is a finite set of states, Σ_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,

²Using 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).

```

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 1: Segment of Java code for a component C2.

and $F_P \subseteq Q_P$ is the set of final states. P defines a regular language $L(P)$. Note that Σ_P contains all provided methods published via the component interface descriptions, e.g., $\Sigma_{P_{C_2}} = \{a, b, c, d\}$ 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 Π (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

$\mathfrak{P} = (S, \Gamma, \hookrightarrow)$ is called *pushdown system* iff S is a set of states, Γ is a finite set of symbols (pushdown alphabet) and $\hookrightarrow \subseteq (S \times \Gamma) \times (S \times \Gamma^*)$ 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

```

1 int C2_x(16);      # pointer into the heap
2 int C2_y(16);
3 module void C2_main(int v0(16), int v1(16)) {
4 int s0(16);
5 C2_main_V:        s0=undef;
6 C2_main_V7:       C2_c_I_V(s0);
7 C2_main_V10:      return; }
8 module void C2_c_I_V(int v0(16)) {
9 int s0(16);
10 int s1(16);
11 C2_c_I_V:         s0=v0, s1=s0;
12 C2_c_I_V1:        s0=1, s1=s0;
13 C2_c_I_V2:        if
14                   :: s1<=s0 -> goto C2_c_I_V9;
15                   :: else -> skip; fi;
16 C2_c_I_V5:        s0=0, s1=s0;
17 C2_c_I_V6:        C2_x=s0, s0=s1;
18 C2_c_I_V9:        return;}

```

Listing 2: Segment of generated SPDS of Listing 1.

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 jMoped³. In Remopla one can declare local loc_q and global $globs$ variables besides the modul parameters (also called procedure parameters) $pars_q \subseteq loc_q$ of a module q . Local variable valuations are stored as bitvectors in the pushdown alphabet.

Let $Vars_q := globs \cup loc_q$, $EXPR_{Vars_q}$ be the arithmetic expressions over these variables and $f^q : Vars_q \rightarrow \mathbb{N}$ a variable valuation. And $\llbracket e_i \rrbracket_{f^q}$ the expression evaluation of the expression $e_i \in EXPR_{Vars_q}$ using the variable valuation f^q . The most important Remopla statements are:

- $\mathbf{x}_1 = \mathbf{e}_1, \mathbf{x}_2 = \mathbf{e}_2, \dots, \mathbf{x}_n = \mathbf{e}_n$; where $x_i \in Vars_q$ is a synchronous parallel configuration change, which assigns each variable x_i the value of $\llbracket e_i \rrbracket_{f^q}$.
- $\mathbf{p}(\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n)$; is a call to module p .
- $\mathbf{return e}$; is a termination of a module while returning the value $\llbracket e \rrbracket_{f^q}$.
- $\mathbf{goto L}$; is an unconditional jump to label L.
- $\mathbf{if :: e}_1 - > \mathbf{s}_1; \mathbf{:: e}_2 - > \mathbf{s}_2; \dots \mathbf{:: e}_n - > \mathbf{s}_n; \mathbf{fi}$; is a conditional statement where $\llbracket e_i \rrbracket_{f^q} \in \{0, 1\}$. One random statement s_k with $\llbracket e_k \rrbracket_{f^q} = 1$ 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

³Integer values are modeled using a bitsize of 16 bits. The keyword *undef* is not defined in Java. It can be simulated through *System.currentTimeMillis()*, because jMoped does not model Java native code into the Remopla model and uses the Remopla statement *undef* instead.

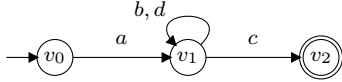


Figure 4: Possible protocol of component C_2 of Figure 3.

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 Moped⁴. 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 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?

⁴But only sequential behavior is considered in this case.

```

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; }

```

Listing 3: Automatically generated initialization code.

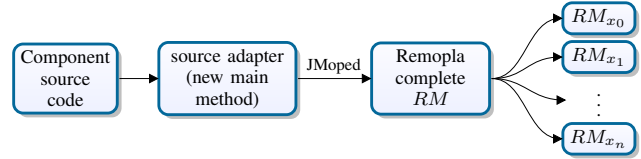


Figure 5: Automatic generation of RM_f (step 2 of Fig. 1).

- 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 jMoped⁵ (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

⁵Otherwise jMoped will drop out syntactical unreachable procedures.

```

1 class C2 { static Object x,y;
2   void main(String s[]) {
3     int i=undef;
4     a(i); b(); c(i); d(); }
5   void a(int q) { ... } ... }

```

Listing 4: Java adapter for RM_f generation of C_2 (Fig. 3).

```

1 int heap[65536];
2 int ptr(16); # heap-pointer to the next free space
3 int C2_x(16); int C2_y(16); # pointer into the heap
4 init jinit; # initialization module
5
6 module void jinit() { ptr=1, # ptr=[1]
7   forall i in (0,65535): heap[i]=0; # heap[*]=[0]
8   C2_x = 0; # C2_x=[0]
9   C2_y = 0; # C2_y=[0]
10  goto C2_a_I_V; }
11
12 module void a(int v0(16)) {# v0=[0,65535]
13   # C2_x=[0], C2_y=[0]
14   int s0(16); # s0=[0,65535]
15   C2_a_I_V: s0 = getRef_C3(); # s0=[23]
16   C2_a_I_V1: C2_x = s0; # C2_x=[23]
17   C2_a_I_V2: s0 = getRef_C2(); # s0=[28]
18   C2_a_I_V3: C2_y = s0; # C2_y=[28]
19   C2_a_I_V4: return; }

```

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

it as the initialization point to get the desired RM_f . An example is shown in Listing 5. Note, in the same sense $C2_main$ was called with uninitialized parameters, now the method a is called with uninitialized parameters.

Once RM_f is constructed, $HalVRE$ is used to evaluate RM_f and to compute on each label (which is a position in the SPDS) all possible data values of the considered variables (see the right side in the comments of Listing 5). An initialization of x is found if the set of possible values of a variable x on a label l contains *only* the value `null` ($Range_l(x) = [0]$) and in a control-flow successor label $l' \in Succ(l)$, it holds that x can be different from `null` ($Range_{l'}(x) \neq [0]$)⁶. After finding such an initialization of x in a procedure f with $l \in f$, f is added to the set $initRef_x$. Thus, we have $f \in initRef_x \Leftrightarrow \exists l', l : l' \in Succ(l) \wedge Range_l(x) = [0] \wedge Range_{l'}(x) \neq [0]$. In line 16 the global variable $C2_x$ is initialized, i.e., the value of $C2_x$ is changed from “[0]” (equivalent to `null`) to “[23]”. See Table I for a complete list of initializations for each component variable of component C_2 (Figure 3).

2) *Calculate Usages*: We search for usages of reference variables in each method separately. Such a use might lead to an unexpected exception during the execution if another method exists setting the used variable to `null`.

Because of the simplicity of the model description language Remopla, the usage information can be calculated using a depth-first search in the syntax tree of the model description. Even when a variable x occurs in an expression (i.e., in a method argument) it is assumed to be used in the current module. Let R be a Remopla model with modules

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` ($0 \in Range_l(x)$) on label l .

method	$initRef$	$usage$	$canNull$
variable x	{ a }	{ b }	{ c }
variable y	{ a }	{ b, d }	\emptyset

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

$M_R = modules(R)$ and labels $L_m = labels(m)$. 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:

$$\begin{aligned}
isused_x(m) &= true \text{ iff } \exists l \in L_m : \exists e \in E_l : x \in e \\
usage_x &= \{m \mid m \in M_R \wedge isused_x(m)\}
\end{aligned}$$

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 $canNull_c$ 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 unconsidered 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

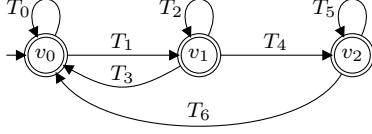


Figure 6: Visualization of the template protocol FSM.

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 $F = Q$ ⁷. 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 *dontCare_x*. It is defined by $dontCare_x \triangleq \Sigma \setminus \{initRef_x \setminus usage_x \setminus canNull_x\}$.

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:

$$\begin{aligned}
 T_0 &\triangleq \{v_0 \xrightarrow{f} v_0 : f \in (canNull_x \cup dontCare_x) \setminus (usage_x \cup initRef_x)\} \\
 T_1 &\triangleq \{v_0 \xrightarrow{f} v_1 : f \in initRef_x \setminus (usage_x \cup canNull_x)\} \\
 T_2 &\triangleq \{v_1 \xrightarrow{f} v_1 : f \in (initRef_x \cup dontCare_x) \setminus usage_x\} \\
 T_3 &\triangleq \{v_1 \xrightarrow{f} v_0 : f \in canNull_x\} \\
 T_4 &\triangleq \{v_1 \xrightarrow{f} v_2 : f \in usage_x \setminus canNull_x\} \\
 T_5 &\triangleq \{v_2 \xrightarrow{f} v_2 : f \in (initRef_x \cup usage_x \cup dontCare_x) \setminus canNull_x\} \\
 T_6 &\triangleq \{v_2 \xrightarrow{f} v_0 : f \in canNull_x\}
 \end{aligned}$$

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 *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 $F = \{v_0\}$.

Theorem 1: $P_{C,x}$ ensures that the reference variable x is not used before it is initialized, nor after it was destroyed.

Proof (Idea): The proof is divided in two steps.

- 1) Only transition rules of T_3 may trigger the usage of x . To perform such a transition rule the state v_1 has to be reached. Every path from v_0 to v_1 requires the call of a safe initialization method (T_1) and setting x to null is not possible.
- 2) After calling a method that uses x (T_4 and T_5), a call to $f \in canNull_x$ (T_6) is leading to the state v_0 . ■

Thus, the $P_{C_2,x}$ ensures the life cycle of a reference variable of a component C . The protocols for the reference variables of the example – $P_{C_2,x}$ and $P_{C_2,y}$ – are shown in Figure 7.

2) *Compute General Component Protocol*: Here the protocols $P_{C,x}$ considering each reference variable $x \in Ref_C$ are intersected. This is done while intersecting the regular languages $L(P_{C,x})$ (cf. [10]). Thus, the component protocol P_C of the component C is computed via $P_C = \bigcap_{x \in Ref_C} P_{C,x}$.

Theorem 2: All unsafe execution traces are excluded by P_{C_2} .

Proof (Idea): As shown in Theorem 1 unsafe usage of each reference variable x is prevented while obeying $P_{C,x}$. While intersecting the corresponding regular languages, all interaction sequences are excluded that are not allowed by all $P_{C,x}$ (where $x \in Ref_C$). ■

The result of the example is shown in Figure 7c. The construction enables us to have several protocols. For example, if there is a protocol defined by a business analyst (via specification, top-down propagation, etc.), a protocol exists defined by the developer and several protocol constructed by the automatic approach that was discussed in this paper, then all these protocols can be summarized using the intersection discussed in this chapter. Thus, our approach does not limit the earlier results, but is modular and extensible. Therefore, it increases safety while considering more properties.

IV. RELATED WORK

A common approach for component composition validation uses behavioral protocols (e.g., [15], [1]). It is a top-down approach where the abstract component behavior is defined manually in general. Moreover, the abstract behavior encodes the contract and is a finite state representation. Similar approaches use e.g., (limited) Petri nets [27] and push-down automata [28].

In our earlier work [3], [2] an improved protocol conformance approach was defined to separate the abstract component behavior from the contract and handle parallelism as well as recursion. There PRS [12] are used. Thus, it is possible to represent an infinite state system, unbounded parallelism, and unbounded recursion. Nevertheless, the model checking is still possible in a practical context [5], [2]. The abstract behavior is generated automatically (bottom-up) from the source code. Hence, although our component

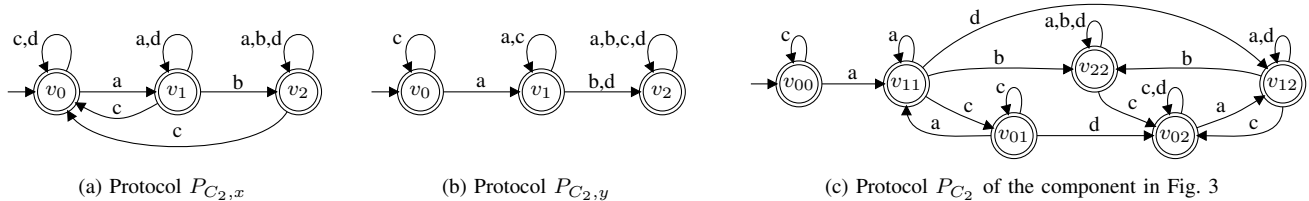


Figure 7: Visualization of protocol $P_{C_2,x}$, $P_{C_2,y}$, and $P_{C_2} = P_{C_2,x} \cap P_{C_2,y}$.

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 interaction sequences (possible bugs) can be prevented automatically. 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 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.

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 configuration 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 component 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 situations 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. However, 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

- [1] S. Bauer, R. Hennicker, and S. Janisch. Behaviour protocols for interacting stateful components with ports. In *Proc. of Int. Workshop on Formal Aspects of Component Software (FACS'09)*. Centrum Wiskunde&Informatica, Amsterdam, 2009.
- [2] A. Both. *Protocol Conformance Checking of Component-based Systems and Service-oriented Architectures*. PhD thesis, Martin Luther University of Halle-Wittenberg, 2010.
- [3] A. Both and W. Zimmermann. Automatic protocol conformance checking of recursive and parallel component-based systems. In *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, volume 5282 of *LNCS*, pages 163–179. Springer, October 2008.
- [4] A. Both and W. Zimmermann. On more predictable implementations of reliable workflows in service-oriented architectures. *IEEE Seventh European Conference on Web Services (ECOWS '09)*, November 2009.
- [5] A. Both and W. Zimmermann. A step towards a more practical protocol conformance checking algorithm. In *35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA 2009)*, pages 458–465. IEEE, 2009.
- [6] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*. NASA Ames Research Center, USA, 2000.
- [7] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11:97 – 111, 2001.
- [8] J. Esparza and S. Schwoon. *A BDD-based model checker for recursive programs*. *LNCS*, 2102:324–336, Springer, 2001.
- [9] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
- [10] J. Hopcroft, R. Motwani, and J. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.
- [11] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.
- [12] R. Mayr. Process rewrite systems. *Information and Computation*, 156(1-2):264–286, 2000.
- [13] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
- [14] J. Obdrzlek. *Model Checking Java Using Pushdown Systems*. LFCS, Division of Informatics, Univ. Edinburgh, 2002.
- [15] F. Plášil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.
- [16] T. Poch and F. Plášil. Extracting behavior specification of components in legacy applications. In *CBSE '09: Proc. of the 12th Int. Symposium on Component-Based Software Engineering*, pages 87–103. Springer, 2009.
- [17] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
- [18] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.
- [19] D. Richter. *Modellreduktionstechniken für symbolische Kellersysteme*. Proc. of the 25. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008.
- [20] D. Richter. *Äquivalenzanalysen - exakt oder nicht - im Vergleich*. Proc. of the 26. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2009.
- [21] D. Richter. Rekursionspräzise Intervallanalysen. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl, 2009.
- [22] D. Richter, R. Kirner, and W. Zimmermann. On undecidability results of real programming languages. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, Maria Taferl, 2009.
- [23] D. Richter and W. Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen*. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.
- [24] D. Richter and W. Zimmermann. Variablenelimination für symbolische Modelle. In *4. Workshop Modellbasiertes Testen im Rahmen der 39. Jahrestagung der Gesellschaft für Informatik*, Lecture Notes in Informatics. Koellen-Verlag, 2009.
- [25] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Dissertation, TU Muenchen, 2002.
- [26] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on moped. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 541–545. Springer, 2005.
- [27] W. M. P. Van der Aalst, K. M. van Hee, and R. A. van der Toorn. Component-based software architectures: a framework based on inheritance of behavior. *Science of Computer Programming*, 42(2-3):129–172, 2002.
- [28] W. Zimmermann and M. Schaarschmidt. Automatic checking of component protocols in component-based systems. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *LNCS*, pages 1–17. Springer, 2006.