TBP: Code-Oriented Component Behavior Specification*

Jan Kofroň, Tomáš Poch, Ondřej Šerý
Charles University in Prague,
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{jan.kofron, tomas.poch, ondrej.sery}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

Abstract

Assuring components compatibility plays a crucial part in developing a reliable component system. Especially, when the components come from different vendors worldwide. In order to do so, an appropriate formalism for behavior specification of components is necessary. We propose a formalism of threaded behavior protocols, *which—unlike most other formalisms—allows for both analysis on the formal level (correctness and substitutability checking) and reasoning about conformance of a specification and the actual implementation. Moreover, the formalism is designed to be simple enough and to directly support constructs known from implementation languages (e.g., method calls, threads, synchronized blocks), so that it is easy to use by a nonprofessional.*

1 Introduction

Over the past years, Component-based Software Engineering (CBSE) has become a well-established paradigm of software development. Via separation of software into reusable building blocks with well-defined interfaces, it facilitates development, deployment, and maintenance of software. Although the idea of constructing new applications solely from the third party off-the-shelf components is still a song of the future, much research and industry effort goes that way.

By moving focus from the development itself to the composition, assuring mutual compatibility of the prefabricated pieces (possibly developed by a number of vendors worldwide) becomes a major challenge. Therefore, the information necessary for assuring compatibility with other components has to be a part of the components interface description as well. Beside method signatures, this includes also a *behavior specification*; i.e., specification of all assumptions the component makes about its environment (e.g., specific call sequencing) on one hand, and guarantees it provides to the environment in the sense of its own behavior (e.g., how the component reacts on a specific method call) on the other hand. Such a behavior specification allows identification of communication errors in a composition of components (e.g., a deadlock and calling a method on a component which is not prepared to accept it). Components are considered compatible if no communication error is found. Formalisms for component behavior specification often support some of the following use cases.

UC1 (*Correctness check*) Given a complete component application, show that it does not contain communication errors [18, 6, 12].

UC2 (*Substitutability check*) Given two components, show that one can be replaced by the other [10, 17, 19]:

- **(a)** in a specific application
- **(b)** in any application

UC3 (*Code conformance*) Ensure that a component implementation conforms to its behavior specification

The use cases UC1 and UC2 help discovering errors in the application design. However, if UC3 is not available, errors can be introduced during

*This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266, and the Q-ImPrESS research project (FP7-215013) by the European Union under the Information and Communication Technologies priority of the Seventh Research Framework Programme.

the implementation itself. Therefore, supporting all the three use cases is important.

During last decades, a number of formalisms for behavior specification of components have been used; they differ in their expressive power, type of notation (textual vs. graphical), level of abstraction, and the set of properties to analyze. However, these formalism are typically too far from the actual code and not designed to support conformance checks against code [5, 2, 1]. Direct support for many concepts a developer is familiar with, e.g., method calls and threads, is often missing.

The goal of this paper is to propose a new formalism for behavior specification of components, *Threaded Behavior Protocols*, which is designed to support UC1 and UC2 on the formal level, on one hand. On the other hand, it is close enough to common implementation languages so that reasoning about specification-to-implementation conformance is possible (UC3). The formalism also directly supports common constructs known from implementation languages (e.g., method calls, threads, synchronized blocks), while being simple enough for use by a common developer.

2 Use Cases

In this section, we discuss the use cases UC1-UC3 in general. The use cases imply several requirements on the target formalism. Later, in Sect. 6, we show how threaded behavior protocols satisfy these requirements.

2.1 UC1: Correctness Check

Let F denote the set of valid specifications in a given formalism. Then, existence of a composition operator $\oplus : F \times F \rightarrow F$, which combines behavior of two specifications, is a natural requirement. The composition operator uses information about bindings between the two components. This information can be present either implicitly (i.e., events with the same names are considered to be bound) or explicitly (i.e., the composition operator can be parametrized by a mapping of event names). In the former case, which is used here, the explicit binding can be realized by an appropriate renaming.

Let a (flat) component application consist of n components and the behavior of the i-th component be described by the specification $P_i \in F$. Behavior of the architecture is then obtained using the composition operator: $P_{arch} = P_1 \oplus P_2 \oplus \ldots \oplus P_n$.

Furthermore, we expect the formalism to provide a mechanism for detection of composition errors. Let us suppose that the mechanism is represented by a predicate $ErrFree(P)$. Correctness of an architecture depends on a specific environment, i.e., it is a relative concept. Therefore, we expect the predicate $ErrFree$ to be defined over closed architectures, i.e., architectures which do not communicate with their environment. Depending on a particular interpretation, $ErrFree$ can reflect various properties, e.g, absence of deadlock.

2.2 UC2: Substitutability

In addition, the formalism is required to support two scenarios of component substitutability checking. The first scenario (UC2-a) is checking that a component can be safely substituted by another one in *a specific* application (environment). For this purpose, the $ErrFree$ predicate is sufficient. The whole modified application is reverified using $ErrFree$. In this case, replacement of a component in an application disregards the original component.

The second scenario (UC2-b) is checking that a component can be safely substituted by another one in *any* application (environment). We denote the *substitutability* relation by $\preccurlyeq_{ErrFree}$. It depends on the $ErrFree$ predicate in the following way:

Definition 1. Let $A, B \in F$. We say that A is *substitutable* for B with respect to the $ErrFree$ predicate, denoted as $A \preccurlyeq_{ErrFree} B$, if $\forall E \in F : ErrFree(E \oplus B) \Rightarrow ErrFree(E \oplus A)$.

Informally, the new component is required to work without composition errors in any environment, where the original component works. This way, substitutability disregards any particular environment.

The substitutability relation can be employed also in the use case UC1. Hierarchical structure of a component application can be exploited to check the application by its levels of nesting. Basically, it suffices to show that (i) every behavior specification of a composite component is substitutable by the behavior specification of the architecture of its subcomponents, and (ii) that the top-level architecture is error free. This is due to the fact that:

$$P_{arch} \preccurlyeq_{ErrFree} P_{frame} \Rightarrow \quad (1)$$

$$\big(ErrFree(P_{Env} \oplus P_{frame}) \Rightarrow \quad (2)$$

$$ErrFree(P_{Env} \oplus P_{arch})\big) \quad (3)$$

Checking the second antecedent (2) is much simpler than checking the consequent (3), as the state space of P_{frame} is usually far smaller that the state space of P_{arch}. The first antecedent (1) does not involve a specific environment. Thus, it suffices to check (1) only once for all applications where the component substitution (update) occurs.

2.3 UC3: Code Conformance

Specification languages and implementation languages are used to capture behavior of an application on different levels of abstraction. While the specification captures only those aspects which are important for the properties of interest, the implementation must capture precisely each step of the computation, including all the technical details unimportant with respect to the observable behavior. In order to bridge this gap, a conformance relation between the implementation and the specification must be established. This relation should extend the notion of substitutability from specification also to code. In other words, we want the relation to hold if the implementation can work correctly in any environment where the specification does.

There are two ways to establish the conformance relation. The first one assumes both implementation and specification created by a user; then, one can use the techniques of model checking or static analysis to show their conformance. The other option assumes having only one of implementation and specification created by a user; in this case, the other one is derived automatically and the conformance relation can be assured by construction.

3 Related Work

Process algebras (e.g., CSP [7] and CSS [11]) are around for a couple of decades. They provide a rigorous, formal way to reason about communicating processes. The semantics of a process is defined in terms of a labeled transition system or a set of traces [4]. These models are checked for satisfying of various notions of compatibility, semantic equivalence (bisimulation relation), and various preorder relations. Also, user-defined properties of models stated in temporal logics can be proved to hold.

These formalisms are very general and do not directly support abstractions used in component based development (interfaces, method calls). Therefore, special languages based on process algebras were proposed for behavior specification of components (e.g., Wright [1] and Darwin [9]). Nonetheless, although they support component abstractions, they often map them to the means of process algebras.

Applying UC1 and UC2 on a process algebra specification is straightforward using the compatibility and preorder definitions already available. However, the specification is still quite far from a real code and conformance of a specification to implementation (UC3) is rarely considered.

The formalism of Interface Automata [5] specifies behavior of a software component directly as a labeled transition system. It defines a notion of optimistic composition and behavioral subtyping, which directly correspond to UC1 and UC2. Again, the abstraction is too coarse and checking the conformance of LTS to an implementation is not considered.

Regarding conformance of a specification to an implementation, a wide range of strategies was investigated. One of them is to let a code model checker cooperate with a tool generating the state space of the specification. Such a strategy is presented in [16]. There, a parallel product of the Java implementation and specification state spaces (synchronized on method calls) is analyzed. Nevertheless, the state space explosion problem is a major obstacle here.

Another option is to restrict the implementation language to bring it closer to the specification. An example of this strategy is implementation of session types [13] in Java presented in [8]. Components implemented in Java communicate over point-to-point sessions. A session type of a session describes what data types and in which order can be transmitted. It is statically verified that the structure of the control flow statements related to an open session corresponds to the structure of the associated session type. To facilitate this verification task, restrictions are put on the implementation language, e.g., preventing aliasing of sessions. Finally, when a session is opened, the runtime checks compatibility of the session types provided by the two communicating components.

Another option to cope with UC3 is to generate code from a specification [15]. In such a case, additional modifications to the generated code are often necessary and can brake the conformance. The last option is to generate a specification from an implementation using user-provided hints. Although, the necessity of user assistance is uncomfortable, this option was chosen as a starting point

for UC3 implementation in this paper. Later, we aim at using the first option, i.e., exploiting a code model checker.

4 Threaded Behavior Protocols

4.1 Syntax

Now, we illustrate the syntax of Threaded Behavior Protocols (TBP) using the example in Fig. 1. The example describes behavior of the `CashDeskLogic` component implementing the business logic of a cash desk in a store. The component is supposed to process individual sales. In order to do so, it keeps the state of the current sale and communicates with other parts of the cash desk (e.g., display and barcode scanner). There are two provided interfaces mediating the input from a keyboard and a barcode scanner, and three required interfaces for accessing a display, a payment component, encapsulating different payment methods (e.g., cash and credit card), and an information system containing inventory of goods.

The basic structure of a component specification in TBP is formed by five parts—declarations of types, declarations of state variables, provisions, reactions on method calls, and threads.

4.1.1 Type Declarations

The *types* section defines enumeration types used in the following sections. The types may be used for declaration of method parameters and state variables.

4.1.2 State Variables

The *vars* section defines state variables influencing behavior of the component. The variables can be accessed only from within the component. Each variable declaration specifies a name, a type, and an initial value. Later on, within the threads and reactions sections, the variables are referenced by assignments and conditions.

There is a special type of variable—*mutex*. A mutex variable serves as a synchronization object, upon which threads can synchronize, e.g., to achieve mutual exclusion.

The `CashDeskLogic` component from the example has a single state variable determining the state of the current sale being processed. The variable is declared on line 9, while its type is declared on lines 3-5.

```
1  component CashDeskLogic {
2    types {
3      states = {INITIALIZED,
4                SALE_STARTED,
5                PAYING},
6      result = {OK, FAILED}
7    }
8
9    vars {states state = INITIALIZED}
10
11   provisions {
12     (
13       ?Keyboard.saleStarted;
14       ?Keyboard.saleFinished
15     )* |
16     ?Scanner.productBarcodeScanned*
17   }
18
19   reactions {
20     Keyboard.saleStarted{
21       if (state == INITIALIZED) {
22         state <- SALE_STARTED
23       }
24     }
25     Scanner.productBarcodeScanned{
26       if (state == SALE_STARTED) {
27         !InformationSystem.getItemPrice;
28         (
29           !Display.productBarcodeNotValid +
30           !Display.changeRunningTotal
31         )
32       }
33     }
34     Keyboard.saleFinished{
35       if (state == SALE_STARTED) {
36         state <- PAYING
37       }
38     }
39   }
40
41   threads {
42     PaymentThread {
43       switch(state){
44         PAYING: {
45           switch(!Payment.pay){
46             OK: {
47               !InformationSystem.recordSale;
48               !Display.thankYouMsg
49             }
50             FAILED: {
51               NULL
52             }
53           };
54           state<-INITIALIZED
55         }
56       }*
57     }
58   }
59 }
```

Figure 1. An TBP specification

4.1.3 Provisions

The *provisions* section declares the intended use of the component in the sense of the calling order of methods on its provided interfaces, the *provided protocol*. The provisions section may contain several provided protocols, each equipped with a list of related methods. The provided protocols can be seen as assumptions, which, when satisfied, result in reactions specified in the *reactions* section. Violation of the protocol is checked and re-

ported as a communication error during verification/simulation. Since the provided protocol does not perform (emit) any events by itself (all events are actually emitted by threads), it can be also seen as an *observer* only checking whether assumptions on the method call order are fulfilled.

The meaning of a provided protocol is a set of traces over an alphabet consisting of method calls. Thus, the grammar of the provision section is inspired by regular expressions. The basic building block of the provision expression is a method call (`?interface.method(`*params*`)`:*returnVal*). The return value is optional. Common regular operators (';', '$*$', '$+$') are used to construct more complex expressions, while parentheses refine the priority of operators. The binary parallel operator ('$|$') stands for interleavings of traces generated by its operands, while the reentracy operator (full '$|?$' and limited '$|x$', where x is the limit) is an unary operator which stands for 'as many parallel executions of the operand as an environment requires'. All choices are external.

The specification in Fig. 1 contains only one provided protocol specified on lines 12-16. The provided protocol specifies that the component can accept the `Keyboard.saleStarted` method followed by the `Keyboard.saleFinished` method. This sequence can be repeated (operator '*'). In parallel, the component can process the `Scanner.productBarcodeScanned` method as many times as the scanner invokes it. The set of methods related to the provided protocol is not specified, which means "default"—all methods on all provided interfaces are monitored.

4.1.4 Reactions

The *reactions* section specifies how a component reacts on a particular method call with respect to its current state. The reaction body consists of method calls on required interfaces (`!InformationSystem.getItemPrice`), assignments to state variables (`state<-INITIALIZED`), `return` statements, control flow statements known from imperative languages (`while`, `if`, and `switch`), and synchronization (`sync(`*mutex*`)`). Beside the expressions over state variables and method parameters, the control flow statements can contain '`?`', which represents a non-deterministic internal choice; i.e., any branch can be executed and the choice cannot be influenced by the component's environment.

For example, in the reaction to the `productBarcodeScanned` method (lines 25-33 in Fig. 1), an information system is queried on a price of an item (`!InformationSystem.getItemPrice`) and either the running total is updated or an error is displayed. The particular value of the running total is abstracted away as it is not important for the behavior.

4.1.5 Threads

With each component, a (possibly empty) set of threads is associated. The number of threads does not change during the computation—no dynamic creation is supported. The threads' task is to issue method calls on component's required interfaces (i.e., on the provided interfaces of other components). Once a thread invokes a method, it switches to the context of the target component, performs the reaction, and finally returns back. Thus, the thread is a source of activity in the model, while reactions are just prescriptions of these activities.

The actions of a thread are described in the same way as those performed within a reaction. An example of a thread specification is on lines 42-58 in Fig. 1.

5 Semantics

In this section, we describe the formal semantics of the TBP models. As already indicated, provisions differ from the imperative parts of the specification (threads and reactions). Not surprisingly, on the formal level, we will capture these two by different means as well. The provisions are represented by a set of important events and a set of allowed traces. To formally capture threads and reactions, another formalism, a variant of a Labeled Transition System enhanced with variables, guards, and assignments—*Labeled Transition System with Assignments* (LTSA)—is used.

Definition 2 (Guards)**.** Let *Var* be a set of variables (for every variable $v \in Var$, we have a finite set of values $domain_v$ which can be assigned to v and an initial value $init_v \in domain_v$). A guard is a finite expression over *Var* derived using the following rules:

- $true$ is a guard,
- $v = l$, where $v \in Var, l \in domain_v$ is a guard,

- $v \neq l$, where $v \in Var, l \in domain_v$ is a guard,
- if X and Y are guards, then $X \wedge Y$ and $X \vee Y$ are also guards.

Note that a mutex m is considered to be a special case of a variable with $domain_m = \{LOCKED, UNLOCKED\}$ and $init_m = UNLOCKED$.

Definition 3 (Labeled Transition System with Assignments)**.** A *Labeled Transition System with Assignments* (LTSA) is a tuple $(S, s_0, F, T, \Sigma, Var)$, where S is a set of states, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, Σ a set of labels, Var a set of variables. Let G be a set of guards over Var, then $T \subseteq S \times G \times \Sigma \times S$ is a transitions relation.

The definition of threaded behavior protocols follows[1]. Informally, it says that a protocol is defined by an alphabet of methods names, a set of variables, a set of provisions (each captured as a set of allowable traces), a set of reactions (represented as LTSA), and a set of active threads (given also as LTSA).

Definition 4. A *threaded behavior protocol* is a five-tuple (Σ, P, R, T, M) or an undefined protocol $\perp$, where:

- $\Sigma = (\Sigma_{all}, \Sigma_{prov}, \Sigma_{req})$ denotes sets of all, provided, and required method names used in the protocol, $\Sigma_{prov} \cap \Sigma_{req} = \emptyset$ and $\Sigma_{prov} \cup \Sigma_{req} \subseteq \Sigma_{all}$. Moreover, only method names from Σ_{all} are allowed in P, R, and T. Where convenient, we use $\Sigma_{ext} = \Sigma_{prov} \cup \Sigma_{req}$ to denote the set of all externally visible method names and $\Sigma_{int} = \Sigma_{all} \backslash \Sigma_{ext}$ for internal only names.
- M is a set of variables.
- P is a set of provisions $\{P_1, P_2, ..., P_n\}$ of a form $P_i = (filter^{P_i}, traces^{P_i})$, where $filter^{P_i} \subseteq (\Sigma_{int} \cup \Sigma_{prov})$ specifies methods observed by the provision and $traces^{P_i}$ specifies a set of allowed finite sequences of events corresponding to methods in $filter^{P_i}$.
- R is a partial function: $(\Sigma_{int} \cup \Sigma_{prov}) \rightarrow LTSA_{(\Sigma_{int} \cup \Sigma_{req}), M}$ representing a mapping of method names to their reactions in a form of LTSA. $LTSA_{\Sigma, M}$ denotes a set of all LTSAs using only methods from Σ and variables from M.
- T is a set of threads $T_1, T_2, ..., T_m$, where $T_i \in LTSA_{(\Sigma_{int} \cup \Sigma_{req}), M}$ is LTSA specifying behavior of the i-th thread.

The definition introduces also an undefined protocol $\perp$, which is used to denote results of invalid compositions.

5.1 Provisions

Basically, each expression within the *provided* section corresponds to a deterministic finite automaton (for a provision P, we denote it by A_P) in a similar way as a regular expression does. To clarify this relation, we describe transformations of non-regular operators that appear in these expressions to obtain a normal regular expression. In addition to regular operators (';' for sequencing, '$+$' for alternative, '$*$' for finite number of repetitions) there are several special (parallel) operators:

- The (and-)parallel operator. '$A \mid B$' yields parallel composition of the expressions A and B, i.e., it generates alternative of all possible interleavings of an event sequence from A with an event sequence from B. For instance, $(a; b) \mid (c; d)$ is equivalent to $(a; b; c; d) + (a; c; b; d) + (a; c; d; b) + (c; d; a; b) + (c; a; d; b) + (c; a; b; d)$
- The or-parallel operator. '$A \mid\mid B$' is equivalent to '$A + B + (A \mid B)$'.
- The limited reentrancy operator. '$A \mid x$' for $x \in \mathbb{N}$ is equivalent to '$A \mid\mid A \mid\mid \ldots \mid\mid A$' where there are x occurrences of A within the expression.

Without any full reentrancy operator, correspondence of a provided expression P to a deterministic finite automaton A_P is straightforward. The full reentrancy operator can be rewritten using the limited reentrancy operator once the composition is complete and the level of parallelism (bounded by the number of threads) is known. The expression '$A \mid?$' can be then substituted by '$A* \mid x$', where x is a constant derived from the number of threads within the specification[2].

[1] For the sake of brevity, we use the term threaded behavior protocol a bit ambiguously to denote both the specification and its semantic model. We believe that there is no risk of confusion.

[2] Note that $A \mid?$ is equal to $A* \mid?$, while $A \mid x$ is not equal to $A* \mid x$ in general.

Later on, during the simulation/verification, A_P is used to verify whether a sequence of events that appears as a consequence of thread execution restricted to the method names present in the $filter^P$ set lies in the language accepted by A_P (i.e., it is allowed by the provision).

5.2 Threads and Reactions

The *threads* and *reactions* section contains description of the actual behavior. In contrast to provisions, which only observe the behavior and signalize errors, threads along with reactions define the behavior itself. Any nondeterminism present in threads and reactions is understood as an internal choice. For this reason, semantics of a thread is LTSA with final states and labels in a form of $a\uparrow$ (method call request) and $a\downarrow$ (method call response), where $a \in \Sigma_{all}$, assignments to state variables, and guards referencing the state variables.

The set of variables *Var* of the LTSA associated with a thread contains all the state variables and mutexes from M. In the case of LTSA representing a method reaction, its *Var* contains also the parameters of the method.

The structure of LTSA is constructed in a bottom-up fashion. The basic building blocks are method calls and variable assignments. The LTSA representing a method call contains three states sequentially connected by two transitions labeled by a method call request and a method call response. A state variable assignment is represented by an LTSA with two states connected by a single transition labeled by the assignment.

LTSAs of a more complex expression is constructed from the LTSAs of its subsexpressions. It is also similar to construction of a nondeterministic finite automaton from a regular expression. The sequence operator (';') corresponds to concatenation of LTSAs, `if` and `switch` correspond to alternative, and the `while` statement is related to repetition. The difference inheres, however, in guards. If the `if` statement contains a condition (not '?'), the corresponding transitions are equipped with the guards derived from the condition and its negation. Similarly, the backward edges coming from the final states of the `while` statement may contain a guard. In the case of nondeterministic choice (`while(?)`, `if(?)`) the guard is always `true`.

Finally, a block synchronized by a mutex `sync(m){...}` adds a new initial state to the LTSA connected by a transition to the original initial state. The new transition is labeled by the guard ensuring that the associated mutex is unlocked `m = UNLOCKED` and by the assignment `m <- LOCKED`, which locks the mutex `m`. The resulting LTSA has only one (newly added) final state with a transition targeting it from each original final state and labeled by the assignment `m <- UNLOCKED`.

5.3 Simulation Semantics

So far, we have presented how the portions of a TBP specification are represented using finite automata and LTSA and roughly how such a representation is constructed from the textual specification. What is to be explained now is how to put these pieces together. Therefore, this section contains a description of how behavior of a software component captured by a TBP specification can be simulated.

Whenever there is a thread calling a method, it is natural to ask about recursion. When simulating TBP, recursion (even indirect) is disallowed[3]. There are two reasons for this. First, we strive for having our models finite. Second, even though recursion is a strong concept on the level of a programming language and can be used inside individual primitive components, we see no use for recursive calls among components.

This restriction allows us to *inline* LTSAs of the reactions into the LTSA of the calling thread (this could be also done on the syntactic level, in a similar vein to the inlining often performed by compilers). Once all the reactions are inlined into the calling threads, the state of a simulation is determined by the valuation of the state variables and mutexes, by the current LTSA state of each thread, and by the current finite automata state of every provision. The initial state of the simulation then corresponds to the valuation which assigns to each variable (or mutex) v its initial value $init_v$, the initial states of LTSA of each thread, and the initial states of each provision automaton.

In a state of the simulation, a transition of LTSA is *enabled* if its guard holds under current valuation of variables. A step of a simulation is then atomic execution of an arbitrary enabled transition of an arbitrary thread. If the chosen transition is labeled by an assignment, valuation of the variables of the next simulation state is modified accordingly. If the chosen transition is a call request

[3]Cyclic bindings among components are allowed as far as they do not result in a recursive call. For example, a callback calls are considered absolutely valid.

or response of a method m, the state of each provision automaton A_P advances iff $m \in filter^P$. When there is a thread in a final state of its LTSA, simulation can make a step in which the thread finishes its execution (the state of its LTSA is fixed and no of its transitions get executed any more).

6 TBP Use Cases

So far, we have focused on a single TBP specification only. Consider now a complete hierarchical application, where each component is associated with a TBP specification. In Sect. 2, the important use cases for analysis of a hierarchical component application were stated. In order to relate them to TBP, the interpretations of the composition operator $\oplus$, the $ErrFree$ predicate, and the substitutability relation $\preccurlyeq_{ErrFree}$ with respect to TBP is discussed in the following.

6.1 Composition

Before defining the composition itself, we first make a simple observation. The names from the sets Σ_{int} and M are not visible to the outer world and thus should not influence the result of the composition. In other words, a protocol defines the same behavior under any arbitrary renaming of Σ_{int} and M. Therefore, without loss of generality, we assume that there are no name clashes in these internal names[4].

Definition 5 (TBP Composition)**.** Let $A = (\Sigma', P', R', T', M')$ and $B = (\Sigma'', P'', R'', T'', M'')$ be threaded behavior protocols. Then:

- if $\Sigma'_{prov} \cap \Sigma''_{prov} = \emptyset$ then $A \oplus B = (\Sigma, P' \cup P'', R' \cup R'', T' \cup T'', M' \cup M'')$, where
  $\Sigma = \big((\Sigma'_{all} \cup \Sigma''_{all}), (\Sigma'_{prov} \cup \Sigma''_{prov}), (\Sigma'_{req} \cup \Sigma''_{req}) \backslash (\Sigma'_{prov} \cup \Sigma''_{prov})\big)$,
- otherwise, $A \oplus B = \bot$.

Moreover, $A \oplus B = \bot$ if either A or B is $\bot$.

In other words, the result of composition is well-defined if the two protocols do not provide a method with the same name, as this would yield a binding of a single required interface to multiple provided interfaces, which is not supported (semantics of such a *broadcast* method call is unclear). For the same reason, the required method names to which a provided counterpart is present in the other component are not added into the resulting Σ_{req}.

[4] Formally, this could be also handled by introducing a name substitution. However, this would obfuscate the otherwise simple definition.

6.2 Correctness Properties

In the following, we present a possible interpretation of the $ErrorFree$ predicate ($BANAFree$) along with a sketch of how to evaluate it algorithmically; i.e., how to perform the verification. Later, we describe a method to decide the substitutability relation based on $BANAFree$.

6.2.1 Checking $BANAFree$

We are basically concerned with communication errors, i.e., violating provisions of a protocol and thus breaking the contract. Two types of provision violations are identified. (i) *Bad activity*, i.e., a thread calls/returns from a method, while there is a provision observing the method, which forbids the call/return at this point. A special case of this error is calling a method which does not have an associated reaction (i.e., calling an unbound required interface). In other words, bad activity is an error resulting from active violation of the provisions, either explicit or implicit. (ii) *No activity*, i.e., all threads have finished, while there is a provision still waiting for a method call to be issued. The automaton representing the provision is not in its final state. The no-activity error represents a situation in which the provisions are violated passively by not finishing a pending work (e.g., not releasing a database cursor).

The $BANAFree$ predicate then holds over those protocols whose behavior may never (under any interleaving of threads) result in a communication error. This is a reachability property; i.e., to decide it, it suffices to visit all reachable states of the simulation of a given TBP (composed of TBP of all application components) as described in Sect. 5.3.

6.2.2 Dealing with Full Reentrancy

It is clear that in presence of the full reentrancy operator, a provision cannot be represented by means of a finite automaton (as the reentrancy operator can be used to express the language of correct bracketing, which is not regular). On the other hand, this does not mean that full reentrancy is

infeasible. In the case of $BANAFree$, the result of the composition is considered to be closed and therefore, the full reentrancy operators can be rewritten using the limited reentrancy operators. The limit is then derived from the total number of threads within the composition.

6.3 Substitutability Relation

The substitutability relation for TBP follows the idea of general substitutability presented in Sect. 2; i.e., TBP is substitutable by another TBP, if the latter one can work without error in any environment where the former can.

Definition 6 (TBP Substitutability)**.** Let $A, B \in TBP$. We say that A is *substitutable* for B, denoted by $A \preccurlyeq B$, if $\forall E \in TBP, \Sigma^E_{ext} \subseteq \Sigma^B_{ext}$: $BANAFree(E \oplus B) \implies BANAFree(E \oplus A)$.

The only unintuitive part of the definition is the restriction put on the alphabet of the environment E, which limits it to $\Sigma^E_{ext} \subseteq \Sigma^B_{ext}$. It says that E can only interact with A by the same methods as with B. In other words, E cannot contain bindings with A in addition to those it has with B. Using additional bindings could introduce new external activity beyond the scope of the parent component thus requiring re-verification of the entire application.

6.3.1 Checking Substitutability

With $A, B \in TBP$, the question to decide is whether $A \preccurlyeq B$ or $A \not\preccurlyeq B$. The basic idea of the algorithm is in traversing both state spaces of A and B using the same external events, simulating the environment and looking for chances to cause an error in communication with A while sustaining error free communication with B. In other words, the algorithm looks for a witness of $A \not\preccurlyeq B$.

While traversing the state spaces, the algorithm maintains sets of states C_A and C_B reachable in A and B using the same external events (i.e., events associated with methods from Σ_{ext}) arbitrarily interleaved with any internal events. In every step, the environment (i) calls a method on A only when B can accept it in *any* state from C_B, (ii) accepts only those method calls which can be issued in *at least one* state from C_B, and (iii) it can decide to finish its activity if this does not imply a no-activity error in *any* state from C_B.

The idea is similar to the idea of the alternating simulation [5]. However, in addition to the bad-activity errors, it is extended to reflect also the no-activity errors.

6.3.2 Dealing with Full Reentrancy

Full reentrancy constitutes a bigger challenge for deciding the substitutability relation than in the case of checking the composition correctness. This is due to the fact that the definition of substitutability quantifies over all environments and therefore no fixed upper bound on the number of threads present in the composition can be assumed.

This problem can be approached in two different ways. First, we can define a thread-limited version of the substitutability relation $\preccurlyeq^x$, which would quantify over only those environments with the number of threads bounded by x. A concrete x could be either estimated or derived from the available information about the target environment.

Second solution would be based on identification of the TBP characteristics that would ensure that, given $A, B \in TBP$, there exists x such that $\forall x' > x : A \preccurlyeq^x B \implies A \preccurlyeq^{x'} B$ and hence $A \preccurlyeq^x B \implies A \preccurlyeq B$. Informally, a point may exist where no new errors might be added by increasing concurrency in the environment. This second approach, however, is just a direction for further research, while the first one is an applicable (brute force) solution.

6.4 Conformance of Java Implementation and TBP

Our approach to ensuring conformance of Java implementation with TBP specification is semi-automatic generation of the TBP specification directly from a Java implementation.

The level of details captured by the implementation is much higher. Therefore, we need to identify the parts important for the behavior on component boundaries and omit the implementation details. However, in general, this behavior can be still too complex for direct translation to TBP[5]. In these cases, we need to introduce over-specification to make the behavior expressible in TBP. In other words, if the specification describes a superset of behavior, and the superset does not imply a communication error in the particular environment, the implementation is also error free. The over-specification may, of course, introduce spurious errors.

[5] In contrast to Java, verification of TBP composition correctness and substitutability are decidable tasks.

Our tool, which is a subject of ongoing work, accepts an annotated Java code. By the annotations, user can define which declarations are important for the behavior on the component's boundary (required methods, provided methods, variables, method parameters) and provide abstraction for types (mapping of Java types to TBP enumeration types). Some pieces of information can be extracted from architecture description, while other ones need a human insight.

The intended usage of the tool is depicted in Fig. 2. After simplification guided by the annotations[6], the Java code is syntactically transformed into TBP. However, the transformation may fail if a construction inexpressible in TBP, say recursion, survives the simplification. In this case, it is up to the user to refine the annotations (or code) to avoid the problem.

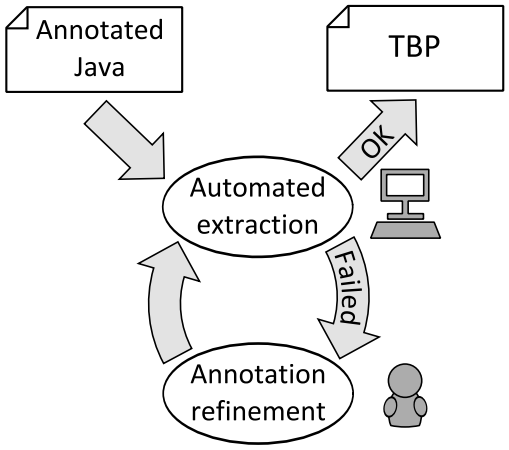

Figure 2. User assisted extraction of TBP specification

This approach is conservative. If the transformation succeeds, the resulting TBP is a valid over-specification of the implementation. We believe, that the annotation refinement should, in most cases, lead to a succeeding transformation. This is based on an observation that the communication among components is relatively straightforward, while complex computations are "hidden" inside components. On the other hand, if the behavior on the component boundaries is intentionally complex, there might be no way to create corresponding TBP, even by hand.

[6] Further details are out of scope of this paper.

7 Future Work

Naturally, the appropriate tool support is vital for any formalism. Although threaded behavior protocols are very general, they put some requirements on the component model—in particular a synchronous communication using method calls is assumed.

The tools are to be developed primarily for the SOFA 2 component model [3], which meets the requirements. We have already started work on tools for analysis, correctness and substitutability checks of threaded behavior protocols (UC1 and UC2), and their semi-automatic extraction from code (UC3). We also plan to employ the Java PathFinder model checker [20] for checking conformance of threaded behavior protocols against the actual Java implementation (UC3) in a similar way as presented in [14].

8 Conclusion

We have presented a formalism for behavior specification of software components—Threaded Behavior Protocols. The strength of the formalism lies in allowing correctness and formal substitutability checks, while being close enough to the implementation languages to make conformance checks against code possible. Moreover, the fact that the formalism directly supports constructs known from the implementation languages (e.g., method calls, threads, synchronized blocks) makes it easier to use by a common developer.

References

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

[2] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes*, 31(2):4, 2006.

[3] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[6] G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International (UK) Ltd., 1985.

[8] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *Proceedings of 22th European Conference on Object-Oriented Programming, Paphos, Cyprus*, 2008.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153. Springer, 1995.

[10] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 692–700. ACM, 1997.

[11] R. Milner. *Communication and Concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[12] M. Moriconi and X. Qian. Correctness and composition of software architectures. *SIGSOFT Software Engineering Notes*, 19(5):164–174, 1994.

[13] M. Neubauer and P. Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*. Springer, 2004.

[14] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 133–141. IEEE Computer Society, 2006.

[15] S. Pavel, J. Noye, P. Poizat, and J.-C. Royer. Java implementation of a component model with explicit symbolic protocols. In *Software Composition*, volume 3628 of *LNCS*. Springer, 2005.

[16] A. Plsek. Extending Java PathFinder with Behavior Protocols. Master's thesis, Charles University in Prague, Czech Republic, 2006.

[17] N. Sharygina, S. Chaki, E. Clarke, and N. Sinha. Dynamic component substitutability analysis. In *FM 2005: Formal Methods*, volume 3582 of *LNCS*. Springer, 2005.

[18] A. Speck, E. Pulvermller, M. Jerger, and B. Franczyk. Component composition validation. *International Journal of Applied Mathematics and Computer Science*, 12(4):581–589, 2002.

[19] I. Černá, P. Vařeková, and B. Zimmerová. Component substitutability via equivalencies of component-interaction automata. *Electronic Notes Theoretical Computer Science*, 182:39–55, 2007.

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.