

# Threaded Behavior Protocols\*

## Technical report

Jan Kofroň, Tomáš Poch, Ondřej Šerý

June 27, 2008

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>

## 1 Introduction

Specification of software component behavior becomes more important with growing complexity of components. A convenient specification can serve for both documentation and verification purposes. Regarding the latter one, one has to choose a proper level of details to capture important behavioral aspects but keep the model small. The aim of the software designer (who is interested in verification of properties of the specification) is to balance the amount of information present in the model and the size of the respective state space to keep the model practically verifiable. Use of a proper formalism makes this task simpler.

### 1.1 Considered components models

**TODO: frame, application, architecture, etc., informally substitutability, error freeness** Having a hierarchical component application, one of the facts we are interested in is whether the involved components communicate correctly, i.e., without errors. Since the components are nested and each composite component acts as a black box to its environment, i.e., the actions happening inside the component are not exposed outside, one could suggest to reason about error freeness of a subtree of the application component hierarchy. On the other hand, the error-freeness of the subtree is a relative concept [1]—in an application the error may occur, while in another not. In other words, the communication correctness is influenced by the context (environment/actual usage) of the component. Therefore, the verification of absence of communication errors makes sense only after the application is entirely specified.

Conversely, the substitutability should not depend on a particular environment. The motivation is the following: If the component developer wants to mark a new version of a component as a replacement of the original (old) one, he/she cannot be aware of particular applications the component is used in, therefore the substitutability should consider specification of these two components only. If this is not feasible, on the other hand, re-verification of the entire application after

---

\*This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838), the Grant Agency of the Czech Republic project 201/06/0770, 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.

component replacement also works. The original component is, however, not referenced anymore, thus the term of substitutability becomes rather misleading.

## 1.2 Component behavioral specification

Recently, we created behavior models of several component Fractal [2] resp. SOFA [3] applications. Our intention was to use model checking to detect communication errors in the models. The models were specified by the formalism of Behavior Protocols [5] resp. Extended Behavior Protocols [4]. We focused on verification of behavioral compatibility of communicating components.

Several issues were identified with respect to the way the developer is forced to think when crafting the specification. The main obstacle is the overall difference between specification and imperative languages the developers are used to. Above all, the trace semantics does not correspond well to the execution semantics of common programming languages. Next, the constructs present in the specification language are not easy to grasp for a developer. These issues often leads to an inappropriate specification either too general or even erroneous—the developers misuse the specification language constructs when trying to model those from an implementation language.

## 1.3 Goals and structure of the paper

Having experience with specification and verification of software component behavior using BP and EBP, we propose a new formalism built upon the original work and explain the motivation for these changes. The proposed formalism is as close as possible to the common imperative programming languages the developers are used to, but remains decidable. The intention is to allow users to express their thoughts by the common means. Moreover, the notion of correctness and behavioral subtyping is formally defined in a modular way to allow reasoning about different properties.

The paper is structured in the following way: In Sect. 2, we present the syntax of the proposed language, while in Sect. 3, we define first the underlying formalism; this is followed by definitions of the most important notions—composability, substitutability, and error-freeness of instances of Threaded Behavior Protocols (TBP). In Sect. 4, we point out the most important parts of contribution of the proposed language. The paper is concluded by Sect. 5 where some directions and ideas for future work are mentioned.

# 2 Syntax

The formalism for specification of software component behavior we propose in this paper is called *Threaded Behavior Protocols* (TBP). From both syntax and semantics point of view, it is based on EBP [4]. However, to keep the paper self-contained, we will describe it root and branch rather than presenting it as a set of changes.

Now, we describe the syntax of Threaded Behavior Protocols (TBP). The basic structure of a component specification in TBP is in Fig. 1.

## 2.1 Types

The *types* section defines enumeration types that are used within the specification. The types may be used for declaration of method parameters and local variables, for instance:

```
states = {CARD_READER_ENABLED, CARD_READER_DISABLED}
```

## 2.2 State variables

The *vars* section defines state variables influencing the component behavior. The variables cannot be accessed from other components (instances). Each variable declaration specifies a name, type, and initial value. Later on, within the behavior section, any value of the same type can be assigned

```

component component_name {
  types {
    types_definition
  }

  vars {
    variable_definition
  }

  provided {
    provided_protocol
  }

  reactions {
    method_reactions
  }

  threads {
    component_threads'_activity
  }
}

```

Figure 1: Basic structure of TBP specification

to the variable, which is an atomic operation. Moreover, the control flow may then depend on the stored value.

There is a special type of variables—*mutex*. A mutex variable serves as a synchronization object, upon which threads can synchronize, e.g., to achieve mutual exclusion. As an example, consider the following variable and mutex definitions:

```

states state = CARD_READER_ENABLED
mutex m

```

## 2.3 Provisions

The *provided* 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 provided protocol specifies how the component should be used by its environment. The provisions section can be seen as the assumption, which, when satisfied, results in reactions (i.e., calls of methods on required interfaces) as specified in the *reactions* section. Violation of the protocol is checked and reported as a communication error during verification/simulation. Since the provided protocol does not perform (emit) any events on itself (all events are actually emitted by threads—see the section about threads), it can be also seen as an *observer* only checking the correctness of the thread’s work. As an example, consider the following provision specification:

```

(?CardReaderCtrl.enable()+?CardReaderCtrl.disable()) |*

```

This snippet specifies that the *enable* or (+) *disable* methods may be called on the *CardReaderCtrl* interface by any number of threads at a time in parallel (|\*).

## 2.4 Reactions

The *reactions* section specifies reactions for particular method calls. For each method on each provided interface, there might be a reaction. The reaction consists of method calls on required interfaces, assignments to the variables, **return**, **while**, **if** and **switch** statements. Once the method is invoked by a thread, the thread performs the actions specified by the reaction. As an example of a reaction specification, consider the following spec snippet:

```

CardReaderCtrl.enable() {
  state <- CARD_READER_ENABLED
}
CardReaderCtrl.disable() {
  state <- CARD_READER_DISABLED
}

```

Here, the specification declares that in reaction to the *enable* method call, the value of the *state* variable is set to *CARD\_READER\_ENABLED* and similarly in the case of the *disable* method.

## 2.5 Threads

With each component, a (possibly empty) set of threads is associated. All threads exist from the beginning, no dynamic creation of threads is supported. The threads' task is to issue method calls on the component required interfaces (i.e., provided interfaces of other components). Thus, a thread is a source of activity in the model. Within the *threads* section, the order in which the threads call required methods and assign values to variables is specified. As an example consider the following thread specification:

```
T1:
  while(?) {
    if (?) {
      !CardReader.PINEntered()
    } else {
      !CardReader.CreditCardScanned()
    }
  }
}
```

The thread specified by this snippet is able to call the *PINEntered* or, alternatively ('+'), the *CreditCardScanned* methods on the *CardReader* interface any finite number of times ('\*').

## 2.6 Syntax in particular sections

As already mentioned, the purpose and character of the provision section is completely different from the other sections. On the other hand, the threads and reaction sections both define behavior in an imperative way. Thus, the grammar of the content of threads and reactions is shared, while the provisions section has its own grammar.

### 2.6.1 Grammar of the provisions section

The grammar of the provision section is inspired by regular expressions. This is motivated by the fact, that the meaning of the provisions section is the set of method call sequences.

The basic building block of the provision expression is a method call.

```
?interface.method(params)
?interface.method(params) : returnValue
```

The method call might contain a return value, when it is important for behavior. Common regular operators (';', '\*', '+') are used to construct more complex expressions, while parentheses refine the priority of operators. The formal definition in EBNF form is in Appendix A.

### 2.6.2 Grammar of imperative parts

Conversely, the imperative parts resembles the imperative languages. Composed statements like while, if and switch are available. However, in order to allow non-deterministic choices, question mark is allowed instead of a condition or expression in the switch statement. There are three basic statements: method call on a required interface (!interface.method(params)), assignment to a state variable (var <- constOrVar or var <- !interface.method(params)) and the return statement(return(constOrVar)).

### 3 Semantics

In this section, we describe meaning of the TBP models whose syntax is mentioned above. After basic definitions, we define the difference between TBP types and TBP instances<sup>1</sup>, which is important for correspondence of TBP with SOFA metamodel. Later, we define the notion of *composition* and *substitutability* and the relation to each other.

**Definition 1** (Guards). Let  $Var$  be a set of variables,  $Val$  a set of values. A guard is a finite expression over  $Var$  and  $Val$  generated by the following rules:

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

**Definition 2** (Labeled Transition System with Assignments). A *Labeled Transition System with Assignments* (LTSA) is a tuple  $(S, s_0, F, T, \Sigma, Var, Val)$ , where  $S$  is a set of states,  $s_0 \in S$  is the initial state,  $F \subseteq S$  is a set of final states,  $\Sigma$  a set of labels,  $Var$  a set of variables,  $Val$  a set of values. Let  $G$  be a set of guards over  $Var$  and  $Val$ . Then  $T \subseteq S \times G \times \Sigma \times S$  is a transitions relation.

**Definition 3.** A *threaded behavior protocol*  $a$  is a five-tuple  $(\Sigma, P, R, T, M)$ , 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  $\Sigma_{all}$  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  $\Sigma_{int} = \Sigma_{all} \setminus \Sigma_{ext}$  for internal only names.
- $M = (M_{var}, M_{mutex})$  is a pair of sets of variables and mutexes representing internal state both being initialized with a value.
- $P$  is a set of provisions  $\{P_1, P_2, \dots, P_n\}$  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:  $(\Sigma_{int} \cup \Sigma_{prov}) \rightarrow LTS_{A(\Sigma_{int} \cup \Sigma_{req}), M}$  representing a mapping of method names to their reactions in a form of LTSA.  $LTS_{A\Sigma, M}$  denotes a set of all LTSAs using only methods from  $\Sigma$  and variables and mutexes from  $M$ .
- $T$  is a set  $T_1, T_2, \dots, T_m$  of threads, where  $T_i \in LTS_{A(\Sigma_{int} \cup \Sigma_{req}), M}$  is a LTSA specifying behavior of the thread.

Moreover, there is an undefined protocol in *TBP* denoted by  $\perp$ .

TODO: komentar k definicim?

#### 3.1 Implicit binding

When designing the behavior specification of a component, the designer uses identifiers of interfaces and method names from the component's frame. In contrast, having the bindings among components reflected as equality of event names (i.e., implicit binding via naming), allows simpler definition of the formalism. Therefore, we distinguish between component types, being the output of the component designer, and component instances, being the result of the renaming operation (implicitly) reflecting the bindings among components.

---

<sup>1</sup>We do not explicitly point out the importance of this as it is beyond the scope of this paper.

## 3.2 Provisions

This section describes the semantics of the expressions present within the *provided* section. As mentioned in the section about syntax, the provided section is intended to specify the set of traces of method calls that may appear on the associated component frame.

Basically, each expression within the *provided* section corresponds to a (nondeterministic) finite automaton 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. ‘ $A \parallel B$ ’ is equivalent to ‘ $A + B + (A \mid B)$ ’.
- The limited reentrancy operator. ‘ $A \mid x$ ’ for  $x \in \mathbb{N}$  is equivalent to ‘ $A \parallel A \parallel \dots \parallel A$ ’ where there are  $x$  occurrences of  $A$  within the expression.
- The (full) reentrancy operator. ‘ $A \mid *$ ’ stands for ‘ $A \parallel A \parallel A \parallel \dots$ ’.

Having defined meaning of all the special operators, the correspondence of a provided expression  $P$  without any full reentrancy operator to a nondeterministic finite automaton  $A_P$  is straightforward. Moreover, 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 ‘ $A * \mid x^2$ ’, where  $x$  is a constant derived from the number of threads within the specification.

Later on, during the simulation/verification,  $A_P$  is used to verify whether the sequence of the events that appear as the 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 provisions).

### 3.2.1 Open questions

Provisions are trace oriented, however, it is desirable to preserve information about thread assignment to the calls. For example, in the provision:  $?a\{?x\}; ?c \mid ?x$ , the desired semantics is that the calls to  $a$ ,  $c$  and both  $x$ ’s can be issued by different threads (even to  $a$  and  $c$ ), while providing that the call to  $a$  can return only after return of the call to  $x$  which was mapped to the first occurrence of  $x$  in the expression. This semantics is desired but neither specified above nor implemented yet. This is due to the fact that preserving the information about thread assignment to calls in provisions would blow up the state space by a number of similarity classes (thread assignment permutations). For optimizing out the state space blow up, we have no satisfactory solution yet.

Just to demonstrate the difference, the following provision:  $P = (?a; ?x; ?b) \mid (?x; ?c)$  is violated by the two threads:  $T_1 = !a; !x; !c$  and  $T_2 = !x; !b$ , if one achieves postponing the call to  $a$  after start of the  $T_2$  call to  $x$  and then postponing  $T_2$  return from  $x$  after  $T_1$  call to  $x$ . Note that the postponing can be achieved using the local variables. When correctly maintaining the thread assignment to provision calls, the erroneous situation would be detected. In contrast, without the mapping, the situation is considered error-free.

Regardless how unlikely and artificial such a specification may look, it is allowed by the formalism and therefore should be targeted in the future by at least detecting the error patterns.

---

<sup>2</sup>Note that  $A \mid *$  is equal to  $A * \mid *$ , while  $A \mid x$  is not equal to  $A * \mid x$  in general.

### 3.3 Threads and reactions

The *thread* and *reaction* section contains description of the autonomous behavior. In contrast to the 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 of the behavior. For this reason, semantic of a thread is LTSA with final states and labels in a form of  $a\uparrow$  and  $a\downarrow$ , where  $a \in \Sigma_{all}$ , assignments to local variables and guards referencing the local variables.

The set of variables  $Var$  of the LTSA associated with a thread contains all variables from  $M$ . In case of LTSA associated with a method,  $Var$  contains parameters of the method.

Regarding the composite statements (**if**, **while**, **switch**) and sequence operator ( $;$ ), the transformation of the behavior description into LTSA is, again, inspired by the transformation of regular expression into finite automaton. The sequence operator ( $;$ ) corresponds to the concatenation of LTSAs, **if** and **switch** corresponds to the alternative and **while** statement is related to the repetition. The difference is, however, in guards. If an alternative statement contains a condition the corresponding transitions are equipped with corresponding guards. Similarly, the backward edge coming from the **while** statement may contain a guard. In case of nondeterministic statements (inner choice - **while**(?), **if**(?)), the guard is always **true**.

Finally, the synchronized block (**sync**( $mtx$ )...) 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 when the transition is used. The action associated with the transition locks the mutex. The result LTSA has just one (newly added) final state. There is an transition coming from each original final state labeled by an action unlocking the mutex.

As the non-deterministic choice is supposed to be internal choice of the component, no determinization is performed.

### 3.4 Composition

Before defining the composition itself, we first make a simple observation. The names from the sets  $\Sigma_{int}$ ,  $M_{var}$ , and  $M_{mutex}$  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 arbitrary renaming of  $\Sigma_{int}$ ,  $M_{var}$ , and  $M_{mutex}$ . To prevent name clashes, the composition operator will use the following substitution:

**Definition 4.** Let  $N$  and  $N'$  be sets of names. Then  $\Theta_{N \rightarrow N'}$  denotes a substitution, which to all names  $n \in N$  assigns a name  $n\Theta_{N \rightarrow N'}$  such that  $n\Theta_{N \rightarrow N'} \notin N'$ . Moreover, the substitution does not assign the same target name to different source names:  $\forall n, m \in N : n\Theta_{N \rightarrow N'} = m\Theta_{N \rightarrow N'} \implies n = m$ .

Using the substitution, the composition operator can be defined as follows:

**Definition 5** (Composition). Let  $a = (\Sigma', P', R', T', M')$ ,  $b = (\Sigma'', P'', R'', T'', M'') \in TBP$ . If:

- $\Sigma'_{prov} \cap \Sigma''_{prov} = \emptyset$

then  $a \oplus b = (\Sigma, P, R, T, M)$ , where:

- $\Sigma = ((\Sigma'_{all} \Theta \cup \Sigma''_{all}), (\Sigma'_{prov} \cup \Sigma''_{prov}), (\Sigma'_{req} \cup \Sigma''_{req}) \setminus (\Sigma'_{prov} \cup \Sigma''_{prov}))$
- $P = P' \Theta \cup P''$
- $R = R' \Theta \cup R''$
- $T = T' \Theta \cup T''$
- $M = (M'_{var} \Theta \cup M''_{var}, M'_{mutex} \Theta \cup M''_{mutex})$
- $\Theta = \Theta_{(\Sigma'_{int} \cup M'_{var} \cup M'_{mutex}) \rightarrow (\Sigma'_{ext} \cup \Sigma''_{all} \cup M''_{var} \cup M''_{mutex})}$

Otherwise,  $a \oplus b = \perp$ . Moreover,  $\forall a \in TBP : a \oplus \perp = \perp$ .

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). The substitution  $\Theta$  is used to prevent collision of internal names from  $\Sigma_{int}$ ,  $M_{var}$ , and  $M_{mutex}$ .

It is worth noting that there are two major options of how to define the composition. First and for the time the favored one is making union of both sets of reactions. The second option is to perform inlining of the reactions of the bound protocols into threads and reactions of the calling protocols. Since the inlining would imply immediate prohibition of recursion of any type, postponing the problem to a later phase (the *ErrFree* predicate) seems to be a reasonable choice.

**Definition 6** (*ErrFree*).  $ErrFree(S)$ , where  $S$  is a TBP specification of a component architecture, is a predicate that holds if there is no communication error inside  $S$ .  $ErrFree(\perp)$  does not hold.

The definition of the *ErrFree* predicate is intentionally very general. In fact, the predicate can be precisely specified for testing of a wide spectrum of properties of component architectures.

### 3.5 Restriction

In order to be able to restrain visibility of certain method names, as in the case of enclosing a component architecture into a frame, we define also a restriction operator over TBP.

**Definition 7.** Let  $a = (\Sigma, P, R, T, V)$ ,  $a \in TBP$  and  $M$  be a set of method names. Then  $a' = R(a, M) = (\Sigma', P, R, T, V)$  is a *restricted* protocol, where  $\Sigma' = (\Sigma_{all}, \Sigma_{prov} \cap M, \Sigma_{req} \cap M)$ .

The following example demonstrates a typical use of the composition and restriction operators.

**Example 1.** Let  $A$ ,  $B$ , and  $C$  be components forming a composite component  $X$  and  $a$ ,  $b$ ,  $c$  their protocols, respectively. Then behavior of the component  $X$  is described by the protocol  $R((a \oplus b \oplus c), M)$ , where  $M$  contains only names of methods appearing on the provided and required interfaces of  $X$ . Note that there is no need to use parentheses and the order of protocols is also irrelevant as the composition operator  $\oplus$  is both commutative and associative.

### 3.6 Substitutability

**TODO: reformulate:** Having a working component application, we can ask, whether its particular component can be replaced by another one. We do not restrict the substitutability to be parametrized by a component environment, since this is no substitutability any more, but rather a question whether a new component will work at a particular place not taking the original component into account. So, the substitutability, to be both reasonable and useful relation, should be defined regardless of the environment:

**Definition 8** (Substitutability). Let  $A, B \in TBP$ . We say that  $A$  is *substitutable* for  $B$ , denoted by  $A \preceq_s B$ , if  $\forall E \in TBP : ErrFree(E \oplus B) \implies ErrFree(R(E, (\Sigma_{prov}^B \cup \Sigma_{req}^B)) \oplus A)$ .

Note, that the substitutability definition is parametrized by particular *ErrFree* predicate.

The ultimate question is whether there is a communication/composition error at the topmost level within the application component hierarchy. Therefore, to make it work, we do not want the frame protocol to be substitutable for its architecture, but, conversely, the component architecture to be substitutable for the component frame. Let us demonstrate this observation by the following example.

**Example 2.** Let  $A$  and  $B$  be specifications of primitive components. Further, let  $A = ?a + ?b$  and  $B = ?a + ?b + ?c$ . Assuming that providing more functionality is not bad, then, obviously,  $B$  is substitutable for  $A$  ( $B \preceq A$ ) but  $A$  is not substitutable for  $B$  ( $A \not\preceq B$ ). In a component hierarchy, to make this work, the only way of arrangement is that  $B$  is the protocol of the architecture and  $A$  is the protocol of the frame, not vice versa, and hence that the component architecture is substitutable for the component frame.

### 3.7 Simulation semantics **TODO: REWRITE**

Within this section we define the simulation semantics of a TBP specification.

#### 3.7.1 Inlining

Having a specification  $a = (\Sigma, P, R, T, M) \in TBP$ , we first perform the *inlining* of the reactions into the thread specification thus obtaining a “complete program”:

**Definition 9** (Parameter substitution). Having a reaction  $i.m(param_1, \dots, param_n)\{q\} \in R$ , we define a substitution of a parameter  $i$  by a value  $val$

**Definition 10** (Inlining). Assuming a specification  $a = (\Sigma, P, R, T, M) \in TBP$ , *inlining* inheres in replacement of all appearances of method calls in the form  $var < -!i.m(val_1, \dots, val_n)$  such that there exist  $r = i.m(param_1, \dots, param_n)\{q\} \in R$  by  $!i.m \uparrow; q[param/val]; ?i.m \downarrow$  in all threads. All method calls are hereby iteratively replaced. Assuming that the inlined specification is denoted by  $a'$ , we write  $I(s) = s'$ .

Note here that recursion (even indirect) has to be disallowed to make the inlining process finite.

Each thread  $t \in T$  of an inlined specification can then execute. That means that it performs the actions from its specification in the order the specification declares, starting from the beginning. The execution of multiple threads is then an interleaving of events of particular threads. More formally, each thread corresponds to LTSA, where transitions are labeled by the events the thread performs. Moreover, a *guard* may be associated with each transition. The transition is allowed to be executed if the guard condition holds.

First, each action (executed event) corresponds to a transition from a state  $a$  to a state  $b$  labeled by the action. The meaning of standard regular operators ( $;$ ,  $+$ ,  $*$ ) is clear and thus we focus on the special operators. Recall now, that each variable and mutex  $v \in M_{var} \cup M_{mutex}$  holds at a particular point a single value. Since the transitions are performed atomically, the value of each variable at each state is uniquely determined by the transition history of all threads. First, we look at the `switch` statement. Its syntax takes the form:

```
switch (v) {
  case val1: prot1
  case val2: prot2
  ...
  case valn: protn
  default: protd
}
```

The `switch` statement corresponds to the alternative of particular `case` branches, however, the branch is not selected nondeterministically, but the one following the case statement with the value that is held by the variable `var` is selected for execution. If there is no such a branch (with respect to the value of the variable), the default branch is selected. Formally, for the  $i$ -th branch, with all transitions from the initial state of the branch there is a guard of the form “ $var = val_i$ ”. The default branch is then guarded by a conjunction of inequalities to values stated in the other branches.

Next, the `while` statement takes the following syntax:

```
while (var == val) {
  prot
}
```

The `while` statement is basically a repetition where the “cycle” transition is allowed if and only if the condition at the `while` keyword holds, that is, if the variable `var` holds the value `val`. In the other case, the transition is not allowed and the `while` statement is accomplished. Again, formally, there is a guard of the form “ $var = val$ ” associated with the “cycle” transition and a guard “ $var \neq val$ ” associated with the ending transitions.

### 3.8 Synchronization model

As mentioned before,  $M$  in a TBP contains a set of state variables  $M_{var}$  and a set of mutexes  $M_{mutex}$ . While the state variables serve for storing the component state, mutexes are used to cope with the synchronization. Mutexes are mainly used to achieve mutual exclusion of execution of multiple threads. To denote mutually exclusive execution of a part of thread actions, the part of the specification is enclosed within the “`sync(mutex)`” statement as follows:

```
sync (mutex) {  
  prot  
}
```

The `sync` keyword is allowed in the *reactions* and *threads* sections of a TBP specification. The meaning with respect to the shape of LTSA is the following: a new state  $s$  is added to the LTSA corresponding to `prot` and a new transition  $t$  leading from  $s$  to the initial state of `prot` is added as well. There is a guard of the form “ $mutex = 0$ ” associated with  $t$  and the label of  $t$  is “ $mutex <- 1$ ”. Since the transitions are executed atomically, the *test-and-set* semantics is hereby assured.

To further simplify the specification construction and to reflect the synchronized methods in e.g., the Java programming language, we define a syntactic abbreviation. The method bodies inside the *reactions* section can be denoted as *synchronized*:

```
sync m1 {  
  reaction  
}
```

The meaning of the specification is the following:

```
m1 {  
  sync (mc) {  
    prot  
  }  
}
```

Where `mc` is a mutex associated with the component instance.

### 3.9 Correctness properties

Recall that the *ErrorFree* predicate has been intentionally defined in a very general way. It is used to decide whether a protocol contains errors or not. Its concrete interpretation may vary, forming thus an extension point of the formalism.

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

#### 3.9.1 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 however 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, as in the second case. (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 does not result in any communication error. In order to decide this, the active behavior of the protocol specified by threads and reactions is simulated along with automatons representing all the provisions.

So far, we are considering just reachability properties.

### 3.9.2 Checking substitutability

Knowing how to evaluate the *ErrorFree* predicate, we can move to the more difficult task of deciding the substitutability relation. Having  $a, b \in TBP$ , the question to decide is whether  $a \preceq b$  or  $a \not\preceq b$ . In fact, it suffices to be able to decide a weaker (implementation-oriented) relation  $\preceq_{impl}$  provided that  $\preceq_{impl} \subseteq \preceq$ , and  $\preceq_{impl}$  is reasonably close to  $\preceq$ . The first requirement implies that any two protocols in the relation  $\preceq_{impl}$  are also in the original relation  $\preceq$ . Therefore, checking only  $\preceq_{impl}$  is correct in a sense, that no unmatching components can be identified as substitutable.

The basic idea of the algorithm is in traversing both state spaces of  $a$  and  $b$  using the same 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  $a \not\preceq b$ . While traversing, the algorithm maintains supersets of states reachable in both  $a$  and  $b$  by using the same observable events from  $\Sigma_{ext}$ .

### 3.9.3 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 always infeasible. For example, in the case of *ErrorFree*, the composition is considered to be closed and therefore, the full reentrancy operators are rewritten using the limited reentrancy operators. The limitation is derived from the total number of threads within the composition.

Full reentrancy constitutes a bigger challenge for deciding the substitutability relation. 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 can be approached in two different ways. First, we can define a thread-limited version of the substitutability relation:  $\preceq^x$ , which would quantify over only those environments with the number of threads lesser or equal to  $x$ . A concrete  $x$  could be either estimated or derived from available information about the concrete environment.

Second, one could think of finding similarity classes that would ensure that  $\exists x$  such that  $\forall x' > x : a \preceq^x b \implies a \preceq^{x'} b$  and hence  $a \preceq^x b \implies a \preceq b$ . This second approach, however, is just an idea for further research, while the first one is an applicable (brute force) solution.

## 4 Conclusion

In this paper, we presented a new specification language for behavior of software components. Having in mind the issues experienced during specification of larger case studies, we based the formalism upon Extended Behavior Protocols, which turned out to be a reasonable specification platform if one is interested in behavioral compliance. Most problems of EBP were caused by almost nonexistent threading and synchronization models. Therefore, we included the thread and mutexes as first-class entities into the proposed language—Threaded Behavior Protocols (TBP).

TBP are closer to code in the following two aspects: (i) The extraction of a part of a model from the component code is simpler due to separation of method reactions from the specification of provisions, and (ii) the verification of code-to-specification compliance has a more straightforward meaning than in the case of (E)BP.

## 5 Future Work

As future work, we plan to simplify a given TBP  $a$  (representing an architecture) to a TBP  $b$  such that  $a \preceq_s b$ . Preferably,  $b$  would be *in a sense minimal* protocol exhibiting the same visible behavior as  $a$ . Probably even  $a \approx_s b$  would be a nice property.

Also, an implementation of a tool performing either or both of substitutability and *ErrFree* verification is to be done.

## References

- [1] J. Adamek and F. Plasil. Erroneous architecture is a relative concept. In *IASTED Conf. on Software Engineering and Applications*, pages 715–720, 2004.
- [2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.
- [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] J. Kofron. *Behavior Protocols Extensions*. PhD thesis, Charles University in Prague, 2007.
- [5] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

## A Grammar

This appendix contains the formal description of the TBP syntax in the EBNF form.

```
tbp = "component", component_name, "{",
      "types", "{", [ { type } ], "}",
      "vars", "{", [ { var } ], "}",
      "provisions", "{", [ { provision } ], "}",
      "reactions", "{", [ { reaction } ], "}",
      "threads", "{", [ { thread } ], "}",
      "}";

component_name = idf;

// Declarations

type = idf, "=", "{", idf, [ { ", idf } ], "};

var = idf idf, "=", idf |
      "mutex", idf ;

provision = [idf] , ['(',method_name_list,')'], "{ p_protocol }";

methodn_name_list = method_name, [ { ", method_name } ]

reaction = [annotation], method_decl, "{", imperative, "};

thread = [idf] , "{", imperative, "};

annotation = "@",idf, ['(',{idf, "=", idf},')'];
```

Provisions grammar:

```
p_protocol = p_alt

p_alt = p_seq, [ { "+", p_seq } ];

p_seq = p_par, [ { ";", p_par } ];

p_par = p_opar, [ { "|", p_opar } ];

p_opar = p_rep, [ { "||", p_rep } ];

p_rep = ap_term, "*" |
        ap_term, "|*" |
        ap_term, "|" {DIGIT} |
        ap_term

ap_term = [annotation], p_term;

p_term = "(" , p_protocol, ")" | p_event;

p_event = "?" method_call, [":", idf] | "NULL"
```

Grammar of imperative parts:

```
imperative = ar_stmt, [ { ";", ar_stmt}];

ar_stmt = [annotation], r_stmt;

r_stmt = r_while |
        r_switch |
        r_if |
        r_synchronized |
        r_event;

r_switch = "switch", "(", value, ")", "{", r_switchbody, "}" |
          "switch", "(", "?", ")", "{", r_nd_switchbody };

r_switchbody = r_case, [ { r_case } ], ["default" , ":", "{", imperative, "}"];
r_case = idf, ":", "{", imperative, "};
```

```

r_nd_switchbody = r_nd_case , [ { r_nd_case } ];
r_nd_case = "case" , ":", "{", imperative, "}";

r_while = "while", "(", cond , ")", "{", imperative, "}";

r_if = "if", "(", cond , ")", "{", imperative , "}" ["else" , "{" , imperative , "}" ];

r_synchronized = "sync", "(", idf, ")", "{", imperative, "}";

r_event = "!", method_call | r_assign | r_return | "NULL" ;

r_assign = idf, "<-" , value;
value = idf | "!", method_call ;

r_return = "return", idf ;

cond = idf, "==", idf | "?" ;

method_name = idf, ".", idf;

method_decl = method_name, "(", parlist_decl, ")"
method_call = method_name, "(", parlist, ")"
parlist_decl = [ idf, idf, [ { ":", idf, idf } ] ];
parlist = [ idf, [ { ":", idf } ] ];

idf = CHAR, [ { DIGIT | CHAR | "_" } ];

```