Under consideration for publication in Formal Aspects of Computing

Threaded Behavior Protocols[1]

Tomáš Poch, Ondřej Šerý, František Plášil, Jan Kofroň

Charles University Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{tomas.poch, ondrej.sery, frantisek.plasil, jan.kofron}@d3s.mff.cuni.cz
http://d3s.mff.cuni.cz

Abstract.

Component-based development is a well-established methodology of software development. Nevertheless, some of the benefits that the component based development offers are often neglected. One of them is modeling and subsequent analysis of component behavior, which can help establish correctness guarantees, such as absence of composition errors and safety of component updates.

We believe that application of component behavior modeling in practice is limited due to huge differences between the behavior modeling languages (e.g., process algebras) and the common implementation languages (e.g., Java). As a result, many concepts of the implementation languages are either very different or completely missing in the behavior modeling languages. As an example, even though behavior modeling languages are practical for modeling and analysis of various message-based protocols, they are not well suited for modeling current component applications, where thread-based parallelism, lock-based synchronization, and nested method calls are the essential building blocks.

With this in mind, we propose a new behavior modeling language for software components, Threaded Behavior Protocols (TBP). At the model level, TBP provides developers with the concepts known from the implementation languages and essential to most component applications. In addition, the theoretical framework of TBP provides a notion of correctness based on absence of communication errors and a refinement relation to verify correctness of hierarchical components. The main asset of TBP formalism is that it links together the notion of threads as used in imperative object oriented languages and the notion of refinement. For instance, this allows reasoning about hierarchical components composed of primitive components implemented in Java without the need of bridging abstractions and simplifications enforced by the modeling languages.

Keywords: Behavior modeling, verification, model checking, refinement, composition, component systems

[1] This work was partially supported by the Grant Agency of the Czech Republic project P202/11/0312 and by the grant SVV-2011-263312.

1. Introduction

The basic idea of component based development (CBD) [21] is to compose complex software from well defined artifacts denoted as components. Individual components are developed independently of each other (possibly by different vendors) and communicate through their interfaces. This independence encourages reuse of a single component in different applications requiring similar functionality.

The application architecture plays a key role in the development process. It identifies the individual components of which the application is composed, their functionality, non-functional properties (e.g., performance) and the way the components communicate with each other—interfaces. Each component defines a set of its provided interfaces (set of methods implementing functionality provided to other components), as well as a set of its required interfaces (functionality the component requires from other components). Each interface has its type, and the types of the connected interfaces must correspond to each other (in the meaning common to object oriented languages).

The benefits mentioned so far are closely related to the fact that component internals are hidden to users. This is referred to as the *black box* view. In particular, only the person who implements a primitive component has access to the source code of the component. Other persons are expected to deal with the component via its interfaces and rely on its correctness.

If a component is used in an architecture, it must be ensured that it conforms to its context in the architecture. In practice, the developers rely on syntactic compatibility of interfaces (at the implementation language level) and a precise documentation. This, obviously, does not guarantee a correct result. A developer may overlook subtle details in the documentation which can lead to erroneous behavior. Also, the documentation does not necessarily reflect the actual component implementation. The errors caused by inconsistency of components can be revealed by testing, or in the worst case, manifest themselves after submission to a customer. This weak point of CBD is addressed by many efforts in academia.

Software verification and CBD can benefit from each other [4]. CBD splits a complex program into smaller fragments—correctness of each primitive component can be verified separately. And afterwards, correctness of composition is to be checked. For such compositional verification, each component is equipped with a formal model specifying its behavior, and when composing components it is checked whether their models fit together.

The requirements posed to a formal model to make it applicable in component-based development are formulated as interface theories in [5]. The requirements include support of incremental design and independent implementability. A selective overview of several diverse approaches related to component behavior evaluation, which ranges from LTS based behavior models, through contracts, analysis of implementation, to summaries can be found in [12].

1.1. Problem statement

Component-based development has found its way into industry. However, the component systems used in practice (e.g. EJB [23], DCOM [16], CCM [25], Koala [30]) do not take advantage of behavior modeling and subsequent analysis.

In our experience based on [31, 1], a part of the problem is that there is no suitable specification language for behavioral modeling aiming at component systems in a comprehensive way. Specifically, no specification language allows writing behavior specification in a straight-forward way, while providing features important for analysis of a component-based system such as behavior composition and refinement. Specifically, crafting behavioral models is a time consuming and error prone task. One of the reasons is that the modeling languages of the formal frameworks are too different from the imperative programming languages used by developers on daily basis.

1.2. Goals

The main objective of this paper is to make the application of behavioral modeling in component-based development more suitable for day-to-day practice.

Based on our experience with specification languages of the Behavior Protocols family, in particular Behavior Protocols (BP) [3, 2] and Extended Behavior Protocols (EBP) [19], we propose the specification

language Threaded Behavior Protocols (TBP) aiming at fulfilling the following two goals. On the one hand, writing specifications in TBP should resemble programming in an imperative programming language. Since programmers are used to the concepts provided by the imperative languages, this aspect should significantly decrease the effort needed to prepare specifications of individual components. On the other hand, the formal framework should provide means for successful application in CBD. This includes a composition operator, as well as refinement relation, both designed with respect to a precisely-defined notion of correctness and imperative concepts of the language. This way, we want to combine the aim of programming languages at practical usability and existing theoretical frameworks focused on analyses ensuring correctness of behavior.

Even though none of the features of the proposed specification language TBP is utterly novel, the contribution of this paper lies in the way these features are put together within the context of a single language. What makes it unique is that the users can reason on refinement of individual components' models, while the computation is driven by threads as it is in a real implementation. Thus, the user can avoid introducing unnecessary specific concepts, "bridging" different abstractions levels and maintain their mappings as the model evolves.

2. Related Work

During the past decades, several languages and formalisms for behavior modeling of software systems have been proposed. They range from very generic ones (e.g., process algebras) to those specific to components (e.g., Darwin [22], Wright [6], Behavior Protocols (BP) [26], BIP [8]). In this section, we focus on those based on labeled transition systems (LTS), as they are well studied, meaning that their properties (e.g., decidability of model checking of particular temporal logic formulas) as well as algorithms for formal reasoning are well established.

2.1. Process Algebras

Classical process algebras (e.g., CSP [18] and CCS [24]) describe a behavior as a set of cooperating processes syntactically defined by a set of recursive equations; each of them associates a process name with an expression determining the process behavior. Formally, the semantics of an expression is given via derivation rules. A number of operators are typically defined to combine elementary actions. The operators include sequencing, alternative, parallel composition typically with the option to create a synchronous product, event hiding, renaming, etc.

When applied in the component context, individual components are represented by separate processes. Synchronization of actions represents issuing a method call on a required interface resp. accepting method call on a provided interface. Names of actions correspond to method names found in the architecture and method parameters and data can be captured in the full calculus by value passing.

A particular example of CSP usage in the context of component systems is the Wright specification language. Wright [6] is an ADL for defining a component-based architecture enriched by behavioral specification. The key abstractions of the component model include a component and its ports (interfaces), and a connector and its roles (interfaces). An assembly is created by binary bindings of ports to roles. Each of the key abstractions is associated with its behavior specification in the form of a process in a subset of CSP.

The process describing the behavior of a component on its ports is called a computation, while the process specifying behavior of a connector on its roles is called glue. Having the behavior of ports, roles, glues and computations specified in CSP, automated checking of composability (based on refinement and deadlock-free testing) is possible. In [29], the approach taken is to transform Wright specifications into plain CSP and use the FDR tool.

Modeling component behavior in a process algebra in principle assumes that every component is an active entity, which does not correspond to the idea of implementation threads "visiting" other components. This leads to the need of fragmenting threads into cooperating processes, or introducing a specific abstractions for component behavior coordination (connectors in [8])

2.2. Automata-Based Languages

Automata based languages define the LTSes of individual communicating systems (with a straight-forward) graphical representation. Such a definition is typically easy to comprehend, since it does not require deep knowledge of the semantics of the given formalism. On the other hand, drawing complex systems tends to be a tedious and time consuming task. Individual formalisms differ in the supported actions (labels), composition operator and by the studied properties and relations over models.

Interface automata introduced by Alfaro and Henzinger [13] distinguish input actions (?name), output actions (!name) and internal actions (name). The parallel composition is used to form more complex systems from simple ones.

Generally, the composition is defined over pairs of composable (having fitting sets of input and output actions) automata A_1 and A_2. A synchronous product automaton $P_A = A_1 \otimes A_2$ is created; A_1 and A_2 synchronize on complementary actions. The result contains error states if an automaton emits an output action and the counterpart automaton is not able to accept it (there is no complementary input action).

Existence of an error state in the product P_A, however, does not mean that A_1 and A_2 cannot work together. If A_1 and A_2 form an open system (i.e., there are still some input and output actions) there can still be an environment E that avoids the error state. If such E exists, A_1 and A_2 are considered *compatible*.

The refinement relation ($\preceq$) supported by the formalism of interface automata preserves the compatibility. In particular, if P and R are compatible and Q $\preceq$ P then also Q and R are compatible. Such notion of refinement can be directly used in the component context to verify hierarchical architectures.

Apart from the interface automata, there are other formalisms, e.g., I/O automata [20], Component Interaction Automata [11] and others.

2.3. UML

Since Unified Modeling Language (UML) [9] is a de-facto industrial standard used for modeling software systems, it is worth mentioning here as well. Although it addresses the problem of behavior modeling, the semantics of related diagrams (activity, state machine, communication, interaction overview, sequence, and timing diagrams) is not defined precisely enough to allow formal verification. Also a formal notion of composition and refinement is missing. Although UML supports profiles refining the semantics for special cases, none of the profiles has been generally accepted yet. Nevertheless, UML is often used to visualize software designs and even for prototype code generation.

3. Application in Development Cycle

Rigorous application of formal methods in software development always requires substantial effort. This includes running analysis tools and interpretation of their results, preparing models in various languages and formalisms as well as maintaining consistency with concurrent activities in the development process. Therefore, to lower the effort and maximize the benefits of using formal methods, it is necessary to carefully articulate their role in the development process.

In comparison to other development paradigms, component based development has the advantage of an explicitly-defined software architecture and component boundaries. Thus, there is a solid ground upon which the formal methods may be built.

Notion of correctness A key benefit is the guarantee that the created software is in a certain sense correct. The notion of correctness may differ depending on the particular formalism and the available tools. Obviously, the formalism cannot provide guarantees related to the software aspects from which it abstracts. In general, three types of correctness are considered (i) composition correctness, (ii) correctness of implementation, and (iii) user-defined properties.

The behavioral model of a component specifies how the component communicates with its environment. This is sometimes referred to as *contract*. In particular, it specifies what the component provides to the environment, under which assumptions, and what it requires from the environment. As the components are composed in a specific architecture, violations of mutual assumptions can emerge, resulting in a composition error. As an example, consider deadlock or invocation of a method on a component not prepared to accept it (e.g., due to omitted initialization).

Another important property is adherence of the implementation of a primitive component, i.e., a component directly implemented in an imperative language such as Java, to its behavior model. This means that, when used in compliance with the assumptions articulated in its behavior model, the component reacts as specified by its behavior model and without low-level implementation errors (e.g., null pointer dereference).

In addition to these issues, some formalisms provide means to specify and analyze violations of user-defined properties. This allows specifying domain-specific properties reflecting the actual business logic of the application. As an example, consider the requirement that all method calls in a component should be preceded by an initialization and eventually followed by a special method call which frees resources (e.g., open files, database cursors, unfinished transactions). User-specified properties are typically provided in the form of a temporal logic formula (e.g., LTL, CTL).

Top-down approach. In the classical waterfall model, the developer is expected to provide the design in an early stage of the development process. If formal methods are applied, certain aspects of the design are expressed formally. For instance, the behavior model is to be provided right after the application architecture is created. When the architecture is hierarchical, the component behavior models can be created level by level—the behavior models of the top level components can be created prior to the architecture of the lower levels of the hierarchy. At each level, the composition of individual models is checked for correctness and also for fulfilling the behavior specification at the higher level. The relation of the behavior composition to the behavior specification at the higher level is referred to as refinement.

When the architecture is ready, the next step is to provide an implementation of the primitive components. Since the behavior model of the primitive components is already available, it is convenient to generate a skeleton of the implementation from the model. Then, the developers are expected to manually implement the aspects of the behavior not captured by the model (e.g., by means of inheritance). Alternatively to skeleton generation, the developers can provide the whole implementation manually. In such case, it is important to find a way to ensure that the implementation conforms to the model of the primitive components. Otherwise, the whole process is not sound.

The top-down approach allows continuous verification as the behavior model is getting more precise. Thus, since errors at higher levels of hierarchy are detected as soon as possible, they can be fixed with small impact on the lower levels.

Bottom-up approach. The top-down approach is not always applicable. An example is analysis of legacy applications. Also in some of the methodologies, which do not follow the waterfall model, rapid prototyping comes first and later the code is refined and improved. In these cases, implementation precedes creation of the behavior model.

The first step is to provide behavior models for all primitive components. It may be either constructed by a skilled reverse engineering specialist or generated from the implementation by means of static analysis or monitoring. Once the behavior models for all primitive components are available, the architecture can be flattened and checked for composition correctness. This may, however, be an infeasible task for current analysis tools (e.g., due to state explosion). In such case, a hierarchy can help to delegate the task to smaller subtasks by checking each composed component individually. In particular, when a behavior specification of the composite components is provided, refinement ensures that the behavior specification corresponds to the composition of primitive components. In the next step, correctness of composition of components at the higher level is checked and so on.

4. TBP Proposal

Based on the comparison of imperative languages with the formalisms presented so far, we can more closely refine the goals.

Goal 1 TBP will feature a straight-forward syntax and semantics to capture the behavior of a component, as well as the assumptions on the behavior of its environment.

(a) We would like the parts of TBP describing (specifying) the behavior of component to be close to imperative programming languages (we have chosen Java as the representative) in both syntax and semantics.

(b) TBP will support specification of assumptions on an environment.

Goal 2 Theoretical framework of TBP will provide guarantees of correctness in the following sense:

(a) The framework will allow comparing actual behavior of a closed system to the expected behavior. Two kinds of assumption violations are to be detected: *bad activity* and *no activity*[2].
(b) Composition operator will reflect the concept of threads.
(c) There will be a preorder refinement relation considering both bad activity and no activity.
(d) The analyses of correctness should be decidable.

To achieve this goal, the theoretical framework of TBP has to precisely define the semantics of concurrent thread execution in a form suitable for definition of correctness and of the refinement relation.

5. TBP Syntax

This section presents the formalism of Threaded Behavior Protocols (TBP) from the user point of view. In particular, intuitive notion of component execution is used to explain meaning of individual concepts. Moreover, since specification of individual components is a typical usage scenario, this chapter, unless explicitly stated, considers just specification of a single component. Note, however, that the formalism of TBP allows expressing syntactically also the result of a composition.

The basic structure of a TBP specification is formed by five parts—declarations of types, declarations of state variables, reactions on method calls, threads, and provisions. While the reactions and threads specify the behavior exercised by the component itself in the imperative manner, provisions specify the behavior of an environment assumed by the component. The assumptions are stated over sequences of provided method calls.

```
component ComponentName {
  types { ... }
  vars { ... }
  provisions { ... }
  reactions { ... }
  threads { ... }
}
```

5.1. Relation to Component Model

The concept of provided and required methods (often formulated using interfaces—groups of methods) is already present in a typical component model. Since TBP is designed to be an extension of such a model, there is no need to provide explicit syntax construction to determine which methods are provided and which methods are required. Later, in the semantics section (Section 7), we assume Σ_{req}, resp. Σ_{prov}, resp. Σ_{int} to denote the set of component's required, provided, and internal methods, respectively. Their content is taken from the component model.

Moreover, the component models define compositions of components via bindings among the interfaces. Thus, since the information about bindings can be taken from the component model, there is no need to specify syntax for composition of TBP specifications.

5.2. Declarations

The *types* section defines enumeration types. The types may be used for declaration of method parameters, state variables, and local variables. The *vars* section contains definition of state variables important for the behavior. The variables can be accessed from within the component (i.e. reaction or thread) only. Later on, within the threads and reactions sections, the variables are referenced by assignments and conditions.

[2] Due to historical reasons we use the terms "no activity" to denote deadlock and "infinite activity" to denote livelock; "bad activity" denotes a situation similar to the error state of Interface automata [13].

```
types {
  result = {OK, FAILED};
  mode = {INIT, RUNNING, MAINTENANCE, SHUTDOWN}
}
```

```
vars {
  mode actualMode = INIT;
}
```

The types fragment contains declaration of two enumeration types. While the `result` type consists of two enumeration values OK and FAILED, the `mode` type consists of four enumeration values. The following example fragment contains declaration of the `actualMode` state variable. The variable's type is `mode` and the initial value is `INIT`.

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

5.3. Reactions

The *reactions* section contains description of the actual behavior performed by the component in reaction to a method call of either a provided or an internal method (a method that is neither provided nor required, but called by a component's thread). Each reaction specified in the section consists of the method name, declaration of arguments and body. The body begins with declaration of local variables followed by the actual behavior.

```
reactions {
  interface.methodName(ArgType1 argName1, ArgType2 argName2, ... ):ReturnType {
    LocalVarType localVar = initialValue;
    ...
  }
```

Local Variables and Arguments Each local variable declaration specifies a name, a type, and an initial value. Arguments, on the other hand, consist just of the type and a name. Local variables and arguments may be accessed only from the reaction body. Moreover, the local variables and arguments are not shared by parallel executions of the body—each execution has its own copy.

Elementary Actions The actual behavior consists of elementary actions composed together by control flow operators. There are three kinds of elementary actions. In the following, `val`, `val1`, `val2`, etc. denote values of defined types. The types of parameters in method calls and assignment correspond to expected types. `Var` denotes a variable.

- Method call — `i.a(val1, val2, ... )`
  The execution of the current reaction is postponed and the `a` method on the `i` interface is invoked with parameters `val1`, `val2`, etc. The `a` method is either required or internal. The execution of the current reaction is resumed when the method `a` is finished.
- Return — `return val`
  Explicit termination of reaction. The returned value may be assigned to a variable at the calling site.
- Variable assignment — `var=val`
  The content of the `var` variable is changed to the `val` value. The variable is either local variable or state variable and the value is either constant (enumeration value), another variable or a method call. In the last case, the return value of the invoked method is assigned to the variable.
- empty action — `NULL`

Control Flow Operators Control flow operators correspond to the control flow statements known from imperative languages. The conditions used in the control flow operators are boolean expression consisting of elementary conditions (equality — `var == val`) connected by common boolean operators (`|| && !`). Moreover, there is the non-deterministic condition denoted by `?`.

Apart from the sequence operator (`;`), there are `if`, `switch` and `while` operators used in the same context as in an imperative language. The `sync` keyword denotes a critical section. It cannot be entered by more than one thread simultaneously. Technically, the mutual exclusion is achieved by atomic test-and-set operation over a variable of the built-in type Mutex (`m` in the fragment).

```
if (condition) {thenBranch}
  else  {elseBranch}
```

```
switch(var){
  case constA: aBranch
  case constB: bBranch
  default: defaultBranch
}
```

```
while(condition){
  loopBody
}
```

```
sync(m){
  criticalSect
}
```

When a non-deterministic condition is used, the executed branch is chosen arbitrarily. The loop statement may be executed any finite number of times (i.e. non-deterministic decision whether to continue or not is taken before each iteration).

5.4. Threads

The *threads* section contains description of the autonomous behavior performed by the component's internal threads. Each thread is declared by a name and a body, which consists of local variable declarations followed by description of behavior. The constructs used to describe reaction bodies are used also to describe thread behavior. Each thread may invoke required methods as well as internal methods. It may change a state variable or a local variable. The number of threads is constant, each thread starts its execution immediately at the beginning of the model execution and once a thread reaches its end it does not perform any action.

5.5. Provisions

Previous sections specify the behavior exercised by the component itself in the imperative manner. In contrast, the purpose of the provisions section is to declare the allowed usage of the component—assumptions posed on the environment (i.e., limit the set of environments with which the component is supposed to cooperate). The key distinction from the imperative part is that the assumptions posed on the environment should be weak. While the imperative parts specify the behavior as precisely as possible, the assumptions must not prohibit too many environments, to enable reuse of the component in different contexts.

Each assumption is specified as a set of allowed sequences of component's provided method calls and returns. The set of allowed environments then includes all environments that do not violate the assumption.

Each provision consists of an expression and a set of methods it constraints. The expression defines a language in the same way as a regular expression. An environment fulfills the provision if all traces it generates, when restricted to the methods from the set, belong to the language.

Elementary expression An elementary expression consists of two actions—a provided method call and the corresponding return. Both actions can contain additional data—either the method call parameters or a return value. Each method used in the expression must be a member of Σ_{Prov}.

```
provisions {
  { i.m(val):retVal } for {i.m}
  { i.n() } for {i.n, i.q}
}
```

The fragment contains two provisions. While the first provision guards all invocations of the method `m` on the interface `i`, the second provision guards invocations of the methods `n` and `q`. The first provision states that the method `m` on the interface `i` is expected to be invoked exactly once by the environment. Moreover, the argument value must be equal to `val` and the method returns `retVal`. The second provision states that the environment calls the method `n` exactly once. The arguments used and the return value may be arbitrary, however. Moreover, the method `q` cannot be invoked by the environment at all.

Operators To construct more complex provisions, the following operators are available.

- Regular operators — sequence (`;`), alternative (`+`) and repetition (`*`)

```
{ i.m(val);i.n()*+i.q() } for {i.m, i.n, i.q}
```

In the example, the environment must call the method `m` with the argument `val` and later it can either call the method `n` arbitrary many times or the method `q` exactly once. In between, before and after

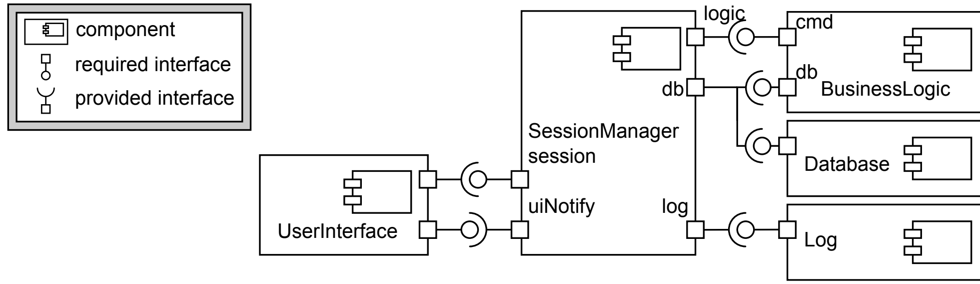


Fig. 1. Architecture employing the SessionManager component

method calls, however, any number of other methods than `i.m`, `i.n`, and `i.q` is allowed, since the other methods are not guarded by the provision.

- Parallel operators — and-parallel(`|`), or-parallel(`||`)
  Parallel operator stands for alternative of all possible interleavings of operands. While for the and-parallel `|` the environment has to follow both operands, for the or-parallel the environment may choose just one operand.
- Reentrancy operator — limited (A `||`n for $n \in \mathbb{N}$) and full (A `||*`)
  A `||`n is equivalent to A `||` A `||` ... `||` A where there are n occurrences of A within the expression. Thus, A may be followed by the environment at most n times in parallel. A `||*` stands for A `||` A `||` A `||` .... Thus, the environment may proceed according to A as many times in parallel as it needs.

```
{ {i.login():ACCESS_DENIED*; i.login():ACCESS_GRANTED; i.n()}|* } for {i.login, i.n}
```

In the example, the environment may attempt to login in parallel. Each attempt, however, must end by a successful login and invocation of the method `n`.

6. Example

6.1. Architecture

Fig. 1 contains an example architecture of a component application. The example contains a fragment of a web-based information system. The application logic is implemented in the BusinesLogic component and the UserInterface component mediates the user input entering the system in form of HTTP requests. The commands from the user, however, do not go directly to the business logic. The communication is intercepted by the SessionManger component that takes care of authentication of commands. SessionManager requires some additional functionality provided by the Log and Database components. The latter one is shared with the BussinessLogic component.

6.2. Behavioral specification for SessionManager

The SessionManager component is expected to work in the environment suggested in Fig 1. In particular, the SessionManager component intercepts the communication between the (web based) user interface and the application logic to provide the authentication feature.

The basic functionality of the component is to associate commands from individual users with sessions. In order to invoke commands, the user interface has to acquire a session id (invoke the `createSession` method provided by SessionManager). Then, if a valid session id is returned, it is used as a parameter for subsequent commands. After SessionManager accepts the command, it checks the session id and passes the command to the business logic. Apart from the main functionality, the component implements a maintenance mode. The mode is automatically turned on when the administrator user is logged in. In the maintenance mode, no new sessions can be created (`createSession` returns `INVALID_SESSION`) and new requests executed in a context of other than the administrator's session causes invalidation of the request session. Moreover, the

session timeout is implemented, so that SessionManager may decide on its own to invalidate an arbitrary session.

It is important to emphasize the purpose of the model since it determines the abstractions to be used. In this case, the goal is to describe dependencies of individual method calls in presence of parallelism. In particular, we want to identify deadlocks and wrong ordering of method calls.

For instance, we do not model the particular authentication algorithm since it is not relevant. Just the result influences the sequencing of method calls and the implementation must be prepared for the positive result as well as for the negative one. Moreover, we do not distinguish individual users. There are just several types of sessions (`ADMIN_SESSION`, `USER_SESSION` and `INVALID_SESSION`). In general, the data abstractions are chosen to reflect the conditions in the code that influence the control flow.

In the model, there is just one provision (lines 16-22) prescribing how the user interface can call the methods on the session interface. In particular, user interface is allowed to submit a command (`invokeCmd`), only if the `createSession` method returns a valid session id (i.e. `USER_SESSION` or `ADMIN_SESSION`). The component is able to process requests from several users in parallel (| and |* operators).

Then, there are reactions for the provided methods `createStatement()` and `invokeStatement()` and the reaction for the internal method `terminate Session()` which is invoked from three different placess. When the user explicitly issues the `CMD_LOGOUT` command (line 46) to invalidate an existing normal user session in the maintenance mode (lines 47-48) and finally on timeout (line 69). The timeout is implemented by the only autonomous thread in the specification, Timer.

```
1  component SessionManager {
2    types {
3      DbResult = {DB_GRANTED, DB_REFUSED};
4      SessionId = {USER_SESSION, INVALID_SESSION, ADMIN_SESSION};
5      UserId = {ADMIN_ID, USER_ID};
6      Command = {CMD_LOGOUT, CMD_OTHER};
7      OperationMode = {NORMAL_MODE, ADMIN_MODE};
8    }
9
10   vars {
11     OperationMode opMode = NORMAL_MODE;
12     Mutex m;
13   }
14
15   provisions {
16     {
17       {session.createSession():USER_SESSION;session.invokeCmd(USER_SESSION,?)*}
18      + {session.createSession():ADMIN_SESSION;session.invokeCmd(ADMIN_SESSION,?)*}
19      + session.createSession():INVALID_SESSION
20     }|*
21     for {session.createSession,session.invokeCmd}
22   }
23
24   reactions {
25     session.createSession(UserId userId):SessionId{
26       DbResult queryResult = DB_REFUSED;
27       queryResult = db.query();
28       sync(m) {
29         if (queryResult == DB_GRANTED) {
30           log.log();
31           if (userId==ADMIN_ID){
32             opMode = ADMIN_MODE;
33             return ADMIN_SESSION;
34           }
35           if (opMode==NORMAL_MODE){
36             return USER_SESSION;
37           }
38         } else { log.log(); }
39         return INVALID_SESSION;
40       }
41     }
42
43     session.invokeCmd(SessionId sessionId, Command cmd)
44     {
```

```
45        if (sessionId == INVALID_SESSION) return
46        if (cmd == CMD_LOGOUT || (sessionId == USER_SESSION && opMode == ADMIN_MODE)){
47          log.log();
48          intr.terminateSession(sessionId);
49        } else {
50          log.log();
51          logic.invokeCmd(cmd);
52        }
53      }
54
55      intr.terminateSession(SessionId sessionId){
56        sync(m){
57          if (sessionId==ADMIN_SESSION){
58            opMode = NORMAL_MODE;
59            log.log();
60          }
61          uiNotify.sessionTerminated(sessionId);
62          log.log();
63        }
64      }
65    }
66
67    threads {
68      Timer {
69        while(?) {intr.terminateSession(?);}
70      }
71    }
72  }
```

7. TBP Semantics

The syntax presented so far accompanied by the information about interfaces from an underlying component model form a TBP specification. Its semantics, as depicted in Fig 2, is defined in two stages.

In model stage, the TBP model is defined. The TBP model is a five-tuple capturing by mathematical means (e.g., Labeled Transition System) the essential information from the TBP specification. Composition is defined at the model stage, so that composition of two TBP models ($\oplus$) yields also a TBP model.

The notion of correctness (i.e. absence of communication errors) and refinement is defined at the LTS stage. The computation of the TBP model is represented by an LTS, either finite or infinite. Communication errors are defined for a closed model (i.e. model which does not exercise any externally observable activity—Σ_{prov} and Σ_{req} are empty) as a property over computation states.

For an open system, refinement is defined as a relation over observation projections. Observation projection is an LTS modeling only externally observable activity (e.g., observation projection of a closed system is a single state with no edges) of the computation. In the special case when the number of external threads expected to use the provided methods of the component is limited to k, the LTS is finite. Finally, notion of refinement is defined as a relation over observation projections.

The motivation behind those stages is to bridge the gap between the TBP specification, which provides relatively rich concepts to the user, and mathematical structures used to clearly define notions of composition, communication errors, and refinement.

7.1. TBP Model

The TBP model precisely defines meaning of the syntax presented so far by mathematical means. As already indicated, provisions differ from the imperative parts of the specification (threads and reactions). Not surprisingly, in the TBP model, those are captured by different means as well. The provisions define a set of important events and a set of allowed traces. To formally capture threads and reactions, a variant of LTS enhanced with variables, guards, and assignments—*Labeled Transition System with Assignments* (LTSA)—is used.

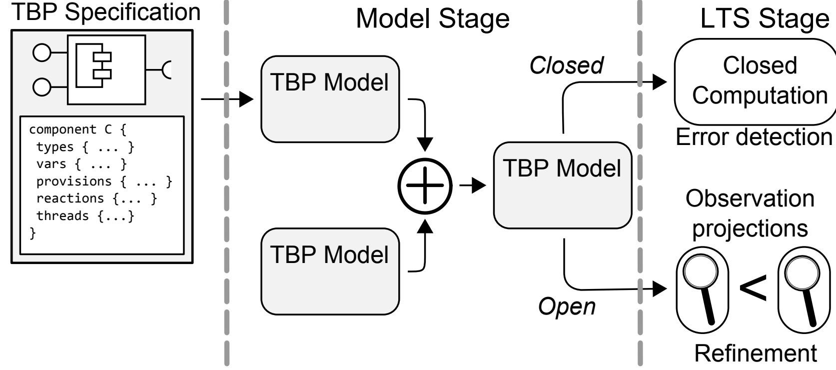


Fig. 2. TBP semantic stages

In the following definitions, let E be a set of enumeration types and V be a set of variables of types from E. Each variable $v \in V$ determines its type and initial value. Dom_e is a set of values of type $e \in E$, Dom_v is a domain of the variable v, $Dom_E = \bigcup_{e \in E} Dom_e$ and $Dom_V = \bigcup_{v \in V} Dom_v$.

A basic element of the transition systems we are going to define is a parameterized transition. It corresponds to a method call, where the method α can contain parameters $v_1, v_2, ..., v_n$:

Definition 1 (Parameterized labels). Let Σ be a set of labels and Par is a set of variables (representing formal parameters). Then, we define the set of parameterized labels $\Sigma_{Par} = \{(\alpha, \langle v_1, v_2, \dots, v_n \rangle) : \alpha \in \Sigma, n \in \mathbb{N} \cup \{0\}, v_i \in Par\}$.

The function $param_i : \Sigma_{Par} \to Par$ returns the i-th parameter of the parameterized label and the function $name : \Sigma_{Par} \to \Sigma$ returns the original label without the parameters.

Labels are later used to model method calls. In particular, parameterized labels allow encoding method parameters as well as return values into labels. Thus, in the following definitions, the set of labels is often parameterized by variables and constants—$\Sigma_{V \cup Dom_E}$.

Definition 2 (Valuation function). The valuation function $\gamma_V : V \cup Dom_E \to Dom_E$ assigns a value to each variable from V. Moreover, it is identity for constants ($\gamma_V(c) = c$ for $c \in Dom_E$). The initial valuation function γ_V^0 assigns initial values to all variables. Modification of the valuation function is denoted as $\gamma_V[v \mapsto c]$.

$$\gamma_V[v \mapsto c](a) = \begin{cases} \gamma_V(a) & : a \neq v \\ c & : a = v \wedge v \in V \\ undefined & : a = v \wedge v \notin V \end{cases}$$

This third case in the definition above denotes the situation when v is not a variable but a constant; obviously, the value of a constant cannot be modified.

Definition 3 (Guards). A guard over V, is a finite expression derived using the following rules:

- $true$ is a guard,
- $v == l$, where $v \in Var$, $l \in Dom_v$ is a guard,
- if X and Y are guards, then $X \wedge Y$, $X \vee Y$ and $\neg X$ are also guards.

The actual value of the guard g for valuation γ_V is denoted as $\gamma_V(g)$ and the set of guards over V is denoted as G_V.

Note that a mutex m is considered to be a special case of a variable such that:

$Dom_m = \{LOCKED, UNLOCKED\}, init_m = UNLOCKED$

Definition 4 (Assignment). Assignment over V is a label in the form

- $v = c$ where $v \in V, c \in Dom_v$
  assigning the constant c of the corresponding type to v

- $v = w$ where $v, w \in V, Dom_v = Dom_w$
  assigning the current value of the variable w to v

  Let A_V be a set of assignments over V.

Definition 5 (Labeled Transition System with Assignments). A *Labeled Transition System with Assignments* (LTSA) is a tuple $(S, s_0, F, \delta, \Sigma, V)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, Σ a set of communication labels, V a set of variables and $\delta \subseteq S \times G_V \times (\Sigma \cup A_V) \times S$ is a transition relation.

Definition 6 (LTSA computation state). Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA. We call the tuple (s, γ_V) a computation state of l if $s \in S$ and γ_V is a valuation function for the set V. The tuple (s_0, γ_V^0) is denoted as initial computation state.

Definition 7 (Enabled LTSA transition). Let $l = (S, s_0, F, \delta, \Sigma, V)$ be an LTSA and $cs = (s, \gamma_V)$ be its computation state. We call the transition $t = (s, g, \alpha, s')$, $t \in \delta$ *enabled* in cs if $\gamma_V(g)$ is evaluated to true.

In the TBP model, LTSAs are used to capture control flow of imperative parts (i.e., reactions and threads). In this context, the set of labels contains parameterized labels representing issuing of a method call (for all required and internal methods).

Let Σ contain method names ($m \in \Sigma$), $\Sigma^{\uparrow}$ contains events for issuing a method call ($m\uparrow \in \Sigma^{\uparrow}$) and $\Sigma^{\downarrow}$ contains events for accepting a method call result ($m\downarrow \in \Sigma^{\downarrow}$). Moreover, $\Sigma^{\uparrow/\downarrow} = \Sigma^{\uparrow} \cup \Sigma^{\downarrow}$. Then, $\Sigma^{\uparrow}_{V \cup Dom_E}$ contains the events for issuing a method call parameterized by all possible combinations of variables from V and constants from Dom_E.

For purposes of TBP model definition, we define $LTSA^{\Sigma}_{V,E}$ to be a set of all LTSAs using labels from $\Sigma^{\uparrow}_{V \cup Dom_E}$ and variables from V. In such case, each transition in δ is thus guarded by $g \in G_V$ and labeled by $l \in \Sigma^{\uparrow}_{V \cup Dom_E} \cup A_V$. Thus, l represents either issuing of a method call (including parameters), or an assignment involving a variable from V. The return value of a method call is assigned to the special purpose *Ret* variable. Thus, to assign the return value to a user specified variable, it suffices to assign the content of the *Ret* variable.

Informally, the TBP model definition says that a model is defined by an alphabet of methods names, a set of variables, a set of provisions (each captured as a set of allowed traces), a set of reactions (represented as LTSA), and a set of active threads (given also as LTSA).

Definition 8 (TBP model). Let E be a set of enumeration types used in a TBP specification. A *TBP model* is a five-tuple (Σ, P, R, T, G), where:

(a) $\Sigma = (\Sigma_{prov}, \Sigma_{req}, \Sigma_{int})$ denotes disjunct sets of provided, required and internal method names used in the model. In addition, where convenient, we use $\Sigma_{ext} = \Sigma_{prov} \cup \Sigma_{req}$ to denote the set of all externally visible method names, $\Sigma_{all} = \Sigma_{ext} \cup \Sigma_{int}$ for all names and $\Sigma_{imp} = \Sigma_{int} \cup \Sigma_{req}$ for names of methods which can be actively invoked in imperative parts.

(b) G is a set of state variables.

(c) P is a set of provisions $\{P_1, P_2, \ldots, P_n\}$ taking the form $P_i = (filter^{P_i}, traces^{P_i})$, where $filter^{P_i} \subseteq (\Sigma_{prov})$ specifies methods observed by the provision and $traces^{P_i}$ specifies a set of allowed finite sequences of events from $(filter^{P_i})^{\uparrow/\downarrow}_{Dom_E}$.

(d) R is a function: $(\Sigma_{int} \cup \Sigma_{prov}) \to (L, \mathbb{N} \to L, LTSA^{\Sigma_{all}}_{G \cup L, E})$ representing a mapping of method names to their local variables (L), a parameter mapping function and reactions in form of LTSA. The L set contains always at least the variable *Ret*.

(e) T is a set of threads $T_1, T_2, ..., T_m$, where $T_i \in (L, LTSA^{\Sigma_{all}}_{G \cup L, E})$ is a tuple specifying a set of local variables and the behavior of the i-th thread in the form of LTSA.

Each provision P_i in (c) represents a set of finite sequences over events representing issuing of a method call and accepting the response. The methods are either provided or internal methods and the events are parameterized by constants from Dom_E. TBP model directly representing a TBP specification states the provisions just over the provided methods. However, after composition, some provided methods become internal methods. The parameters of response events represent return values. This allows reflecting return values in sequences and, in particular, describing an environment which behaves with respect to the value

obtained from the method call. Notice, that events are not parameterized by variables as in other cases, but just by constants from Dom_E (actual values of method calls).

The labels in the LTSA for reactions in (d) contains calls of methods from Σ_{imp} and assignments over local variables and state variables. In addition, constants from Dom_E are allowed in method calls. The *Ret* variable is a special purpose variable used to store results from method calls. The parameter mapping function is used to assign parameters from a parameterized method call event α to variables from L.

7.1.1. From TBP Specification to TBP Model

Method sets (Σ), global variables (G), and filters ($filter^{P_i}$) directly correspond to the sets from TBP specifications. Provisions ($traces^{P_i}$) and construction of LTSAs (R,T), however, deserve precise definitions. Formally, we define the function $model(TBPSpec, k)$ taking a TBP specification and the number of threads from the environment allowed to enter the specification in parallel as arguments.

Provisions Every set $traces^{P_i}$ is represented by a finite state machine (FSM). Its construction directly follows the algorithm for construction of FSM from a regular expression. In particular, method calls are translated into a sequence of two events representing issuing a method call and return from a method call. Both events are parameterized—either by parameters or by return values. The parallel operator results in an interleaving of FSMs induced by its operands. Finally, the reentrancy operator is treated as a sequence of or-parallel operators. The number of parallel operators is given by the parameter k of the *model* function.

Imperative Parts The structure of $LTSA^{\Sigma}_{V,E}$ 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 and, optionally, a Ret value assignment. A state variable assignment is represented by an LTSA with two states connected by a single transition labeled by the assignment.

The LTSA of a more complex expression is constructed from the LTSAs of its subexpressions. It is also similar to construction of a nondeterministic finite automaton from a regular expression. The sequence operator (';') corresponds to the 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 deterministic condition (not '?'), the corresponding transitions are equipped with the guards derived from the condition and its negation. Similarly, the edges coming from the final states of the `while` statement may contain a guard.

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`.

7.1.2. Composition

Before defining the composition itself, we first make a simple observation. The names from the sets Σ_{int} and G 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 G. Therefore, without loss of generality, we assume that there are no name clashes in these internal names[3].

Moreover, when two models are being composed, they should 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. Thus, to capture these requirements, we define notion of composable models.

Definition 9 (Composable models). Let $A = (\Sigma', P', R', T', G')$ and $B = (\Sigma'', P'', R'', T'', G'')$ be TBP models. We say, that A and B are composable iff

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

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

Composition of two composable TBP models is again a TBP model. The composition makes a union of the corresponding sets of provisions, reactions, threads, and state variables.

Definition 10 (TBP Composition). Let $A = (\Sigma', P', R', T', G')$ and $B = (\Sigma'', P'', R'', T'', G'')$ be composable TBP models. Then:

$$A \oplus B = ((\Sigma_{prov}, \Sigma_{req}, \Sigma_{int}), P' \cup P'', R' \cup R'', T' \cup T'', G' \cup G'')$$

where
$\Sigma_{prov} = \Sigma'_{prov} \cup \Sigma''_{prov}$,
$\Sigma_{req} = (\Sigma'_{req} \cup \Sigma''_{req}) \setminus (\Sigma'_{prov} \cup \Sigma''_{prov})$ and
$\Sigma_{int} = \Sigma'_{int} \cup \Sigma''_{int}$

Notice that the set of methods provided by composition is a union of methods provided by the original models. This way, a method provided by the input model which is at the same time required by the other input model stays in the set of provided methods in the composition. Thus a provided method can be required (thus, invoked) by several components composed together by sequential application of the composition operator.

Definition 11 (Closed TBP model). Let $M = (\Sigma, P, R, T, G)$ be a TBP model. We call M closed if $\Sigma_{req} = \emptyset$

The essence of a closed model is that it does not communicate with the environment. The closed computation introduced in the following text considers only the threads from T as a source of activity and does not expect the environment to invoke any method.

7.2. Analysis of Closed Models

In this section, we define the computation of a closed TBP model as a finite LTS, called *a closed computation*. Intuitively, a closed computation is created by composition of LTSAs of individual threads and reactions. In particular, the way individual LTSAs are put together is inspired by a typical stack-based execution model of imperative languages.

The number of threads is fixed in the closed TBP model. There is a single stack for each thread. The top of a stack refers to the actual position in the LTSA being currently executed by the thread. Also, the local variables and parameters referenced by LTSA guards and assignments are on the stack. Thus, each *computation state* is represented by a number of stacks and valuation of state variables.

Definition 12 (Computation state). A *Computation state* of the TBP model (Σ, P, R, T, G) is a tuple $(Stacks, \gamma_G)$, where $Stacks$ is a (multi)set of stacks—sequences of tuples (s, γ_L). The size of $Stacks$ corresponds to the number of threads ($|Stacks| = |T|$), γ_G and γ_L are valuation functions and s is a state of an LTSA ℓ. The LTSA ℓ either captures behavior of a reaction or a thread from the model.

A *computation transition* represents an atomic change of the computation state. Such a change is either a modification of a stack or modification of a state variable or both. A change of the stack size corresponds either to issuing a method call or accepting a method call response. Those transitions are labeled by corresponding parameterized labels. The data in those labels are the actual values used in the particular method calls and returns. Thus, if the method names are from Σ, then labels are from $\Sigma^{\uparrow/\downarrow}_{Dom_E}$. Since the labels in LTSA may be also parameterized by variables, *a parameter valuation function* is defined to get the actual values of these variables.

Definition 13 (Parameter valuation function). Let $\gamma_V : V \cup Dom_E \to Dom_E$ be a valuation function. Then, we define the parameter valuation function $\gamma^{\Sigma}_V : \Sigma_{V \cup Dom_E} \to \Sigma_{Dom_E}$ in the following way:
$\gamma^{\Sigma}_V((\alpha, < p_1, p_2, \ldots, p_n >)) = (\alpha, < \gamma_V(p_1), \gamma_V(p_2), \ldots, \gamma_V(p_n) >)$.

In the following definition, a stack is treated as a pair $(top, tail)$ or *null*, where *top* is the item on the top of the stack, *tail* is the rest of the sequence and *null* represents an empty stack.

Definition 14 (Computation transition). Let $l = (S, s_0, F, \delta, \Sigma^{\uparrow}_{V \cup Dom_E}, G \cup L)$ be the LTSA corresponding to the top of the active stack ($s \in S$, L is the set of variables considered by valuation γ_L). Let $(((s, \gamma_L), tail) \cup Stacks, \gamma_G)$ be a computation state of the TBP model (Σ, P, R, T, G), where $((s, \gamma_L), tail)$ is

$$\text{(a)}\quad \frac{\delta{:}s \xrightarrow{g,m^{\uparrow}<a_{1..n}>} s',\gamma_{G\cup L}(g){=}true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{m^{\uparrow}<\gamma_{G\cup L}(a_{1..n})>} (\{((s',\gamma_L),tail)\}\cup Stacks,\gamma_G)}$$

where $m \in \Sigma_{req}$

$$\text{(b)}\quad \frac{\delta{:}s \xrightarrow{g,v=e} s',\gamma_{G\cup L}(g){=}true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{\tau} (\{((s',\gamma_L[v{\mapsto}\gamma_{G\cup L}(e)]),tail)\}\cup Stacks,\gamma_G[v{\mapsto}\gamma_{G\cup L}(e)])}$$

where $v \in G \cup L$
$e \in G \cup L \cup Dom(v)$

$$\text{(c)}\quad \frac{\delta{:}s \xrightarrow{g,m^{\uparrow}<a_{1..n}>} s',\gamma_{G\cup L}(g){=}true}{(\{((s,\gamma_L),tail)\}\cup Stacks,\gamma_G) \xrightarrow{m^{\uparrow}<\gamma_{G\cup L}(a_{1..n})>} (\{((s'_0,\gamma^0_{L'}[p(i){\mapsto}\gamma_{G\cup L}(a_i)]),((s',\gamma_L),tail))\}\cup Stacks,\gamma_G)}$$

$\forall i \in \{1..n\}$ and $m \in \Sigma_{prov} \cup \Sigma_{int}$
$R(m) = (L', p, l')$ where L' is a set of local variables, $p : \mathbb{N} \to L'$ is a parameter mapping function, and $l' = (S', s'_0, F', \delta', \Sigma_{G\cup L\cup Dom_E}, G \cup L')$

$$\text{(d)}\quad \frac{s{\in}F}{(\{((s,\gamma_L),((s',\gamma_{L'}),tail))\}\cup Stacks,\gamma_G) \xrightarrow{m^{\downarrow}<\gamma_L(Ret)>} (\{((s',\gamma_{L'}[Ret{\mapsto}\gamma_L(Ret)]),tail)\}\cup Stacks,\gamma_G)}$$

Fig. 3. Rewriting rules defining a *computation transition*

the stack of the thread producing the transition. Moreover, $m < a_{1..n} >$ is a shorthand for $(m, < a_1, \ldots, a_n >)$ and $m < \gamma_V(a_{1..n}) >$ for $(m, < \gamma_V(a_1), \gamma_V(a_2), \ldots, \gamma_V(a_n) >)$. The transitions among computation states are defined by the rewriting rules in Fig. 3. While the top part of the rule references the LTSA capturing control flow of a method performed by a thread, the bottom part defines a new transition.

In Fig. 3, the rule (a) represents invocation of a required method. The computation transition is labeled by the method identifier parameterized by the actual values of its arguments. The rule (b) represents an assignment. Just the valuation of the given variable is changed and the computation transition is labeled by the silent action. The rule (c) represents invocation of a provided or internal method. In either case, a method reaction is to be executed by the thread. Thus, the target state of the computation transition has a new item at the top of the active stack containing the initial state of the invoked method's reaction. The valuation of local variables contains correct values for the method's arguments. The rule (d) represents the final step of a method reaction. Since the final state of the corresponding LTSA has been reached ($s \in F$), the stack in the target state of the computation transition is popped and the content of the special purpose variable *Ret* (return value) is copied one level higher on the stack. The computation transition is labeled by the return method identifier parameterized by the actual value of the *Ret* variable.

Put together, computation states and transitions form a closed computation:

Definition 15 (Closed computation). Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model. Then the *closed computation* of M is the tuple $C(M) = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$, where:

- Each label from $(\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E}$ is an event representing either issuing or acceptance of method calls parameterized by constants from Dom_E.
- $s_0 = (Stacks_{init}, v^0_G)$ is initial computation state of the model. The set $Stacks_{init}$ contains a stack for each thread $t = (L, LTSA_t)$ from T containing a single item (s^0_t, γ^0_L), where s^0_t is the initial state of $LTSA_t$ and γ^0_L is the initial valuation of L.
- $\delta \subseteq S \times (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\} \times S$ is a transition relation corresponding to the computation transition
- S is a set of computation states reachable by δ from s_0

- $F \subseteq S$ is a set of final states. The state s is final if for each thread t its stack contains just single item (s, L) such that s is a final state of $LTSA_t$.

The closed computation of a closed TBP model is finite as long as the model does not contain (even indirect) recursive calls in reactions. To keep the properties we are interested in decidable, we consider only finite TBP models in the following text. Technically, we prohibit recursion. We believe this is not a huge harm from the practical point of view, since we consider recursive calls among components to be a bad practice. Practically, a tool can detect a possible recursion and refuse to provide any results in such case. Moreover, since there are no required methods, there are no transitions of type (1) (Definition 14)

The definitions presented so far provide us with the precise meaning of a set of TBP specifications composed together such that they form a closed TBP model. The semantics in this case is given in the form of finite LTS, which provide straightforward definition of communication errors in the following sections.

7.2.1. Communication Error

The formalism of TBP distinguishes two kinds of communication errors. There are inherent errors, which appear in consequence of a closed computation, and errors with respect to provisions.

Definition 16. Let $C(M) = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a closed computation of a closed TBP model M.

There is *inherently no activity* in a state $s \in S$ if $s \notin F$ and there are no transitions leading from s. The set of all states with inherently no activity is denoted as $F^{\oslash}$.

There is *infinite activity* in state $s \in S$ if there is no path from s to a final state or to a state where inherently no activity is. The set of all states with infinite activity is denoted as F^{∞}.

There is *internal infinite activity error* in state $s \in S$ if there is infinite activity in s and all paths leaving the state s contain only τ events. The set of all states with internal infinite activity error is denoted as F^{∞}_{int}. Apparently, $F^{\infty}_{int} \subseteq F^{\infty}$.

No activity denotes the situation when a thread gets stuck in a state waiting for a guard which never starts to hold (e.g., to enter a critical section created by `sync` keyword or waiting for a certain value of a variable). In such a case, the thread is waiting for an action to be performed by another thread (leave the critical section, set the variable), which does not occur. Since such a situation is clearly undesirable, such a state is considered to be erroneous.

On the other hand, the infinite activity is desirable in some cases—especially those emphasizing reactive nature of a system while not modeling shutdown at all. The special case of infinite activity—internal infinite activity, however, is undesirable, since there the computation performs only internal actions with no effect observable by other components. For instance, internal infinite activity captures the situation, when a thread is actively waiting (in a loop) for a guard.

We consider as erroneous the states from $F^{\oslash}$ and F^{∞}_{int}, while F^{∞} can be desirable in some models. Additionally, there is another class of errors induced by the provisions.

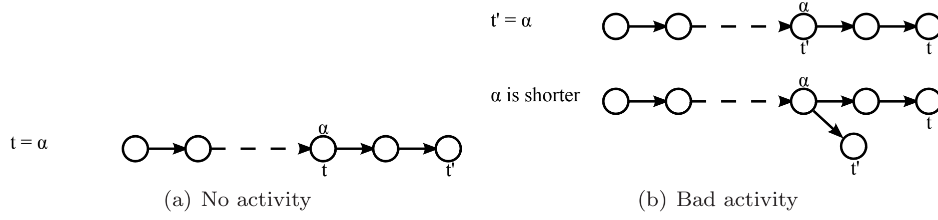
Provisions of individual components express assumptions posed on the environment in the form of traces. As the models are being composed together, the particular environment of the component is being formed. Once the system is closed, it is checked whether all provisions are obeyed.

Since provisions are based on traces, we define traces generated by closed TBP model first.

Definition 17 (Computation trace). Let $C = ((\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a closed computation of a closed TBP model. Then, we call the finite sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$, $\alpha_i \in (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}$ *computation trace*, if there is a sequence of computation states $s_0, s_1, \ldots, s_{n+1}$ such that $\forall i, 0 \leq i < n : \wedge(s_i, \alpha_i, s_{i+1}) \in \delta$.

Moreover, we call the computation trace *terminating* when $s_{n+1} \in F$, *stuck* when $s_{n+1} \in F^{\oslash}$, *diverging* when $s_{n+1} \in F^{\infty}$ and *internally diverging* if $s_{n+1} \in F^{\infty}_{int}$.

The set of terminating computation traces is denoted as $L(C)^{\checkmark}$ and stuck computation traces as $L(C)^{\oslash}$. $L(C)^{\infty}$ contains the set of representatives of diverging computation traces—all diverging traces sharing the same (already diverging) prefix are represented just by the prefix. The set of representatives of internally diverging computation traces is denoted as $L(C)^{\infty}_{int}$. $L(C) = L(C)^{\checkmark} \cup L(C)^{\oslash} \cup L(C)^{\infty}$.



(a) No activity (b) Bad activity

Fig. 4. Examples of provision violation

The set $L(C)$ characterizes a closed computation. It contains all traces leading to successful termination as well as traces leading to no activity and traces leading to diverging states. Having such characterizing set of computation traces, we can define the provision violation.

Definition 18 (Trace restriction). Let $t = \alpha_0, \ldots, \alpha_n$ be a computation trace consisting of parametrized labels $\alpha_i \in (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}$ and $f \subseteq \Sigma_{imp}$ be a set of labels referred to as a filter. Then, the trace restricted by f, $t \upharpoonright f = \alpha'_0, \ldots, \alpha'_m$ is a computation trace consisting of $\alpha'_i \in f^{\uparrow/\downarrow}_{Dom_E}$ such that

$$t \upharpoonright f = \begin{cases} \alpha_0.(t_1 \upharpoonright f) & : name(\alpha_0) \in f \\ t_1 \upharpoonright f & : otherwise \end{cases}$$

where $t_1 = \alpha_1, \ldots, \alpha_n$ is the trace t without the first parametrized label and . is the concatenation operator. Moreover, restriction of the empty trace is the empty trace.

Informally, the trace restriction operator removes from the trace the parametrized labels that are not based on a label belonging to the filter. As an example, consider the following trestriction:
$m_1(v_{11}, v_{12}, ..., v_{1n}); m_2(v_{21}, v_{22}, ..., v_{2n}); m_3(v_{31}, v_{32}, ..., v_{3n}); m_4(v_{41}, v_{42}, ..., v_{4n}) \upharpoonright \{m_1, m_4\} =$
$m_1(v_{11}, v_{12}, ..., v_{1n}); m_4(v_{41}, v_{42}, ..., v_{4n})$.

Definition 19 (Provision violation). Let (Σ, P, R, T, M) be a closed TBP model and C its closed computation. We say, that the provision $P_i \in P$, $P_i = (filter^{P_i}, traces^{P_i})$ is obeyed by a trace $t \in L(C)$ iff $t \upharpoonright filter^{P_i} \in traces^{P_i}$. The provision is not violated in the TBP model if it is obeyed by all the computation traces from $L(C)$. In the other cases, we say that the provision is *violated*.

With violation of provision defined in general, let us discuss specific kinds of violations. Violation of a provision P_i for some i occurs, if there is a trace in L(C) such that its restriction $t \notin traces^{P_i}$. In such case, let $t' \in traces^{P_i}$ be a trace, sharing the longest prefix α with t.

Fig. 4(a) illustrates the case when $t = \alpha$, which is denoted as *no activity*. In such case, the restricted trace t follows the provision up to the certain point (α) and then immediately terminates without following the rest of the trace t'. It corresponds to the situation when the model is expected to perform some action (e.g., close a session), however it does not and just terminates, instead.

Definition 20 (No activity). A TBP model with a closed computation C generating computation traces L(C) and containing a provision (filter, traces) contains *no activity error* if there is $t \in L(C) \upharpoonright filter$, $t \notin traces$ such that it is a prefix of a trace $t' \in traces$.

Fig. 4(b) reflects the remaining two situations. Either $t' = \alpha$ or α is shorter than both t and t'. In the former case, the trace t is a witness of a situation when a source model attempts to invoke a provided method of a target model, which is already in a final state and does not expect any further method calls. The latter case, on the other hand, reflects the situation when the target component does not expect the invoked method, but expects another method call.

Definition 21 (Bad activity). The TBP model with closed computation C generating computation traces L(C) and containing provision (filter, traces) contains *bad activity error* if there is $t \in L(C) \upharpoonright filter$, $t \notin traces$ such that it is not a prefix of any trace $t' \in traces$.

For the purposes of the following text, we define the predicate *ErrFree* over the set of all closed TBP models.

Definition 22 (ErrFree). Let M be a closed TBP model. Then we define

- $ErrFree_{BA}(M) \Leftrightarrow$ there is no bad activity in the model.
- $ErrFree_{\oslash}(M) \Leftrightarrow$ there is neither inherently no activity nor no activity in the model.
- $ErrFree_{\infty_{int}}(M) \Leftrightarrow$ there is no internal infinit activity in the model
- $Errfree(M) \Leftrightarrow ErrFree_{BA}(M) \wedge ErrFree_{\oslash}(M) \wedge ErrFree_{\infty_{int}}(M)$

Assuming C is a closed computation of M, $Errfree(M)$ can be alternatively defined as $(L(C)^{\checkmark} \upharpoonright filter) \subseteq traces \wedge (F^{\oslash} = F^{\infty}_{int} = \emptyset)$

7.3. Analysis of Open Models

So far, a notion of correctness for closed TBP models was presented. In the context of hierarchical component models, however, developers often deal with open systems. Thus, it is necessary to extend the notion to the realm of open TBP models.

Even if there is an error (in the sense of previous paragraphs) in the open system, it often depends on the particular environment whether the error becomes evident or not. The environment may steer the open system away from the error so that the error is never reached in the closed system that results from a composition.

Thus, instead of mere identification of errors in an open system, comparison of open systems with respect to an environment is preferred. In particular, the question is whether a TBP model I behaves correctly in all environments where a TBP model S behaves correctly. Such relation is referred to as refinement in the rest of this paper.

In a typical scenario, I represents a complex model (I stands for the implementation, which is typically a composition of other models) while S is a relatively simple model capturing only the important aspects of behavior with respect to communication with the environment.

Having the refinement relation, the hierarchical system verification is divided into two subtasks. The first task is done when an open system is being created. The developer of a composite component provides the model S and ensures that the actual component behavior corresponds to S—by means of refinement. The second task is done when the open system is put into a particular environment to create either a closed system or an open system on the higher level. In former case, the developer uses the specification S to verify correctness of the closed model by means of the *ErrFree* predicate. In the latter case, the specification S is put together with other components to form the implementation of the composed component. Then, the refinement is checked again on the higher level.

Formally, the refinement is defined as follows:

Definition 23 (Refinement). Let I and S be TBP models and E be the set of all TBP models such that $\forall e \in E : e$ is composable with both I and S, and $e \oplus I$ is a closed TBP model.

- We say that I refines S with respect to bad activity iff
  $\forall e \in E : ErrFree_{BA}(e \oplus S) \Rightarrow ErrFree_{BA}(e \oplus I)$
- We say that I refines S with respect to no activity iff
  $\forall e \in E : ErrFree_{\oslash}(e \oplus S) \Rightarrow ErrFree_{\oslash}(e \oplus I)$
- Finally, we say that I refines S iff
  $\forall e \in E : ErrFree(e \oplus S) \Rightarrow ErrFree(e \oplus I)$

Definition of the refinement is based on the errors considered. Moreover, the implication in these definitions ensures transitivity of all three refinement relations as stated in the following lemma[4].

Lemma 1 (Transitivity). Let A, B and C be TBP models such that A refines B and B refines C. Then, A refines C.

The rest of this section provides a means for deciding whether I refines S. This is done in several steps.

First, an open TBP model is transformed into provision-driven computation. It is LTS similar to the closed computation, however it also contains the transitions labeled by input actions representing the actions the

[4] The proofs can be found in [27].

environment is allowed (or expected) to perform. Those input actions are distinguished from the actions actively performed by the model (output actions).

In the next step, an observation projection is created from the provision-driven computation. The purpose of the observation projection is to resolve the non-determinism in the model. It is a pessimistic approximation of the provision-driven computation representing the behavior of the model as observed by an environment. In particular, the environment is not able to distinguish states of the model reached by the same sequence of observable actions. All these states are represented by a single state (super-state) in the observation projection. Moreover, the super state allows the environment to perform only the actions allowed by a state of provision-driven computation it represents. On the other hand, the observation projection requires the environment to be ready for all actions that may occur in any state the super state represents.

The final step when deciding whether I refines S is parameterized alternation simulation. Basically, the alternation simulation [13] identifies pairs of states which must fulfill a property in order to ensure refinement of specifications. The property P parameterizing the particular variant of refinement is designed to preserve the corresponding error.

Currently, the refinement requires that the number of threads originated in the environment is limited. The limit k goes through all the following definitions and theorems. In this sense, we define a weaker notion of refinement as follows:

Definition 24 (Refinement up to k threads). Let I and S be TBP models and E is a set of all TBP models such that $\forall e \in E : e \oplus I$ is a closed TBP model and e does not invoke more than k provided methods of I in parallel.

- We say that I refines S with respect to bad activity up to k threads iff $\forall e \in E : ErrFree_{BA}(e \oplus S) \Rightarrow ErrFree_{BA}(e \oplus I)$
- We say that I refines S with respect to no activity up to k threads iff $\forall e \in E : ErrFree_{\oslash}(e \oplus S) \Rightarrow ErrFree_{\oslash}(e \oplus I)$
- Finally, we say that I refines S up to k threads iff $\forall e \in E : ErrFree(e \oplus S) \Rightarrow ErrFree(e \oplus I)$

7.3.1. Provision-driven Computation

In contrast to the closed specification, the open specification, besides its internal threads, allows threads from the environment to invoke its provided methods. The way the external threads invoke the provided methods is, however, limited by provisions. Thus, in the provision-driven computation (which is an LTS) also externally triggered activity occurs (i.e. transitions representing invocation of a provided method by an external thread).

Formally, provision-driven computation is a tuple $C_{PD} = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$. Notice that it differs from the computation signature by the set of labels appearing on transitions—while the computation transition uses only Σ_{imp}, there can be also input actions representing provided methods (Σ_{all}). In the remainder of this paper, we use ?m to denote $m \in \Sigma_{prov}$, !m to denote $m \in \Sigma_{req}$ and finally τm to denote $m \in \Sigma_{int}$.

In contrast to closed computation, the number of stacks in the individual states changes to reflect the external threads allowed by provisions to call the provided methods. Moreover, individual states also contain the information about the actual state of provisions.

The first step to create a provision-driven computation is to combine individual provisions of the specification. This is done in two phases. First, each provision is normalized to formally guard all methods from Σ_{prov}. The set of traces allowed by the normalized provision is equal to the set of traces allowed by the original provision. The normalization is done by interleaving all original traces by arbitrary invocations of methods that were not guarded by the original provision. In the second phase, the intersection of normalized provisions is created to achieve combined provision.

Definition 25 (Normalized provision for k threads). Let $P = (filter, traces)$ be a set of provisions of a TBP model M and Σ_{prov} the set of provided method names of M. The normalized provision for k threads is defined as $P^{norm,k} = (\Sigma_{prov}, traces | L((m_1 + m_2 + \ldots + m_n) * ||k))$, where the operator $|$ produces all interleavings of its operands, $m_1, \ldots, m_n \in \Sigma_{prov} \setminus filter$ and $L((m_1 + m_2 + \ldots + m_n) * ||k)$ is a language of traces capturing at most k parallel repetitive invocations of methods $m_1, \ldots, m_n$.

Definition 26 (Combined provisions for k threads). Let $P = \{P_1, P_2, \ldots, P_n\}$ be a set of provisions of a TBP model M where $P_i = (filter^{P_i}, traces^{P_i})$. Then, the combined provisions of M is a provision $Prov^k_M = (\Sigma_{prov}, \bigcap_{i=1,\ldots,n} traces^{norm,k}_{P_i})$ where $traces^{norm,k}_{P_i}$ is the set of traces of the normalized provision $P_i^{norm,k}$.

If the provisions are seen as finite automata, which is possible as each of them defines a regular language, the combined provision is a finite automaton formed as the intersection of automata corresponding to particular normalized provisions.

As long as the environment features less then k threads, the composed provisions allow the environment to perform exactly the same behavior as was allowed by the original set of provisions. The fact is expressed by the following lemma:

Lemma 2. Let $M = (\Sigma, P, R, T, G)$ be a closed TBP model and $N = (\Sigma, \{(\Sigma_{prov}, Prov^k_M)\}, R, T, G)$ is the same model where the set of provisions P was replaced by composed provisions of M for k threads where k is the number of threads in M ($|T|$). Then, $ErrFree(M) \Leftrightarrow ErrFree(N)$.

The Lemma 2 can be proven in the following way. Using the alternative definition of ErrFree(M) ($(L(C)^{\checkmark} \upharpoonright filter) \subseteq traces \wedge (F^{\oslash} = F^{\infty}_{int} = \emptyset)$), one can see that the provisions influence only the $(L(C)^{\checkmark} \upharpoonright filter) \subseteq traces$ part of definition. Moreover, the closed computation $L(C)$ remains the same for both M and N. For a particular provision P_i of M, let us denote $M_{errtr} = (L(C)^{\checkmark} \upharpoonright filter_{P_i}) \setminus traces_{P_i}$ and $N_{errtr} = L(C) \setminus \bigcap_{i=1,\ldots,n} traces^{norm,k}_{P_i}$ and prove $M_{errtr} = \emptyset \Leftrightarrow N_{errtr} = \emptyset$. There is a trace t in M_{errtr} iff $\exists t' \in L(C)$ such that $t = t' \upharpoonright filter_{P_i}$ and at the same time $t \notin traces_{P_i}$. Either t is too short (it is a prefix of a trace from $traces_{P_i}$) or α is the first symbol of t that differs from the trace from $traces_{P_i}$. In the former case, t' is just a prefix of a trace from $traces^{norm,k}_{P_i}$ (we were adding interleavings, thus, no trace was shortened). In the latter case, since $\alpha \in filter_{P_i}$, one can find the same (i-th) occurence of α in t' as well and there is no prefix of a trace from $traces^{norm,k}_{P_i}$ that would follow t' up to and inluding α (we were adding just arbitrary interleavings of methods not belonging to $filter_{P_i}$). Thus, $t' \in L(C) \setminus \bigcap_{j=1,\ldots,n} traces^{norm,k}_{P_j}$.

For the opposite direction, the idea is the same, one just have to use the fact that number of threads is limited by k, thus there is no problem with limited reentrancy in normalized provisions.

Once composed provisions representing the behavior that the model M expects from the environment is available, provision-driven computation can be constructed. The following definitions are based on definitions of closed computation. For instance, the provision-driven computation state is a computation state enriched by a position in the combined provisions.

Definition 27 (Provision-driven state for k threads). Let $M = (\Sigma, P, R, T, G)$ be a TBP model and $Prov^k_M = (S, s_0, \delta, F)$ its composed provisions for k threads. A *state of Provision-driven computation of M for k threads* is a tuple $(Stacks, \gamma_G, s)$, such that the tuple $(Stacks, \gamma_G)$ forms computation state and s identifies a state from $Prov^k_M$ ($s \in S$).

In addition to the transition of a closed computation, the provision-driven computation transition is changing the state of combined provisions as a method is invoked. Moreover, there are transitions representing invocation of provided methods by threads from the environment.

Definition 28 (Provision-driven transition for k threads). The rewriting rules in Fig. 5 define the transitions among provision-driven states. While the top part of each rule references the computation transition (Def. 14), the bottom part defines a new provision-driven transition. Let $(Stacks, \gamma_G, s_{Prov})$ be a provision-driven computation state of the TBP model $M = (\Sigma, P, R, T, G)$, where s_{Prov} is a state of $Prov^k_M = (S_{Prov}, s_{0Prov}, \delta_{Prov}, F_{Prov})$.

In Fig. 5, the rule (a) produces a transition which does not modify the provision state. The rule (b) and (c) produces a transition representing active invocation of a method by the model. The provision state is modified to reflect the invocation. If the provision does not provide the required transition, the computation state causes bad activity error—$(Stacks, \gamma_G, s_{prov}) \in F^{BA}$. The rule (d) produces a transition representing invocation of a method by environment. The provision state is modified to reflect the invocation.

It is worth to notice that while the transitions defined in (a), (b), and (c) represent an activity performed actively by the model (!m), the rule (d) states that the model is allowed to perform the activity if asked by the environment (?m).

(a) $$\frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G)}{(Stacks,\gamma_G,s_{prov})\xrightarrow{\alpha}_{PD}(Stacks',\gamma'_G,s_{prov})}$$

where $\alpha = \tau$ or α is a required method call

(b)(c) $$\frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G),\delta_{prov}:s_{prov}\xrightarrow{\alpha}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\xrightarrow{\tau}_{PG}(Stacks',\gamma'_G,s'_{prov})} \quad \frac{(Stacks,\gamma_G)\xrightarrow{\alpha}(Stacks',\gamma'_G),\delta_{prov}:s_{prov}\not\xrightarrow{\alpha}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\in F^{BA}}$$

where α is a provided or internal method call

(d) $$\frac{\delta_{prov}:s_{prov}\xrightarrow{m^{\uparrow}<a_{1..n}>}s'_{prov}}{(Stacks,\gamma_G,s_{prov})\xrightarrow{m^{\uparrow}<a_{1..n}>PG}(Stacks\cup\{((s'_0,\gamma'_L),null)\},\gamma_G,s'_{prov})}$$

where $R(m) = (L', p, l')$
s'_0 is the initial state of l'
$\gamma'_L = \gamma^0_{L'}[p(i) \mapsto a_i]$

Fig. 5. Rewriting rules defining *provision-driven transition for k threads*

Definition 29 (Provision-driven computation for k threads). The provision-driven computation of a TBP model $M = (\Sigma, P, R, T, G)$ for k threads is the tuple $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F, !F)$, where:

- A label from $(\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E}$ is either an event representing issuing of a method call or acceptance of a method call parameterized by constants from Dom_E.
- $s_0 = (Stacks_{init}, v^0_G, s^0_{Prov})$ is initial provision-driven state. The set $Stacks_{init}$ contains a stack for each thread $t = (L, LTSA_t)$ from T containing single item (s^0_t, γ^0_L), where s^0_t is the initial state of $LTSA_t$ and γ^0_L is initial valuation of L. s^0_{Prov} is an initial state of $Prov^k_M$.
- $\delta \subseteq S \times (\Sigma_{imp})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\} \times S$ is a transition relation corresponding to the provision-driven transition for k threads.
- S is a set of provision-driven computation states reachable by δ from s_0
- $!F \subseteq S$ is a set of imperative final states. The state $s = (Stacks, \gamma_G,\ s_{Prov})$ is a final imperative state if for each model's thread t there is a $stack \in Stacks$ containing just single item (s', L) such that s' is a final state of $LTSA_t$. There are no other stacks in *Stacks* representing the threads originated in the environment.
- $F \subseteq S$ is a set of final states. The state $s = (Stacks, \gamma_G, s_{Prov})$ is a final state if it is imperative final state and s_{Prov} is a final state of $Prov^k_M$.

In addition to the final states F representing the situation when both internal threads and provisions are in a final state, we also define the set $!F \subseteq S$ to denote the states where all threads are in a final state, but there is no restriction on the provision state ($!F \subseteq F$). Those states reflect the situation when the model is neither obliged to perform any action on its own nor to terminate.

Similarly to the closed computation, using the definition of provision-driven final states, we define the sets $F^{\oslash}$, F^{∞} and F^{∞}_{int}. Moreover, we define F^{BA} to be a set of states causing the bad activity error (Definition 28 (c). In contrast to the closed computation, mere existence of an error state (e.g. $s \in F^{\oslash}$) does not automatically mean that the model is useless. It can still work perfectly in a number of environments which use it in a way such that the error is avoided.

The provision-driven computation is well defined even for closed systems. Such computation features no transitions labeled by input actions (?m). Moreover, it can be used for detection of error states.

Lemma 3. Let M be a closed TBP model containing k threads and $C^k_{PD}(M)$ is its provision driven computation for k threads. The set of error states F^{BA} is empty iff there is no bad activity state in M.

7.3.2. Observation Projection

A key step when deciding whether the I model (implementation) refines the S model (specification) is to compare their ability to work in various enclosing environments—the model I has to work in all environments where S works. Thus, only the model's behavior observable by the environment is important. The environment cannot make any assumptions regarding the internal non-determinism of the model (including internal communication). To include this fact in further reasoning about refinement, we define the *observation projection* of the provision-driven computation. The observation projection is an LTS which abstracts from the non-determinism of the original computation in a pessimistic way—whenever an error can occur due to the non-determinism, it must be reflected in the observation projection.

Each state of the observation projection (super-state) represents a set of states of the original provision-driven computation. The individual states represented by the same super-state cannot be distinguished by any environment. For instance, states originally connected by a τ transition always belong to the same super-state.

Let $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a provision-driven computation of a TBP model M for k threads. Let $A \subseteq S$ be a set of states. We define τ-*closure*(A) as to be a set of states reachable from $s \in A$ by a set of externally invisible transitions—τ-transitions originated as assignments (cases (a) and (b) in Definition 29).

Definition 30 (Observation projection for k threads). Let $C^k_{PD}(M) = ((\Sigma_{all})^{\uparrow/\downarrow}_{Dom_E} \cup \{\tau\}, S, s_0, \delta, F)$ be a provision-driven computation of TBP model M for k threads. Then the *observation projection of M for k threads* is a tuple $C^k_{OP}(M) = ((\Sigma_{ext})^{\uparrow/\downarrow}_{Dom_E}, S_{OP}, s^0_{OP}, \delta_{OP}, F_{OP}, !F_{OP})$ such that

- $S_{OP} \subseteq 2^S$
- $s^0_{OP} = \tau closure(\{s_0\})$
- $\delta_{OP} : S_{OP} \times (\Sigma_{ext})^{\uparrow/\downarrow}_{Dom_E} \to S_{OP}$
  $\delta_{OP}(s_{OP}, ?m) = \tau closure(s'_{OP}) \Leftrightarrow \forall s \in s_{OP} \exists s' \in s'_{OP} : (s, ?m, s') \in \delta$
  $\delta_{OP}(s_{OP}, !m) = \tau closure(s'_{OP}) \Leftrightarrow \exists s \in s_{OP} \exists s' \in s'_{OP} : (s, !m, s') \in \delta$
- $F_{OP} = \{s_{OP} : \forall s \in s_{OP} : s \in F\}$
- $!F_{OP} = \{s_{OP} : \exists s \in s_{OP} : s \in !F\}$

Moreover, we define sets of various error states. In particular

- $E^{BA}_{OP} = \{s_{OP} : \exists s \in s_{OP} : s \in F^{BA}\}$
- $E^{NA}_{OP} = \{s_{OP} : \exists s \in s_{OP} : F^{\oslash} \cup F^{\infty}_{int}\}$
- $E_{OP} = E^{BA}_{OP} \cup E^{NA}_{OP}$

Observation projection simplifies the original provision-driven computation by reducing non-determinism. The internal choices are expected to behave as in the worst case w.r.t. bad activity; if one of the states belonging to a super-state (e.g., several states connected by internal actions) produces an output action, it must be produced also by the super-state. On the other hand, an input action leaving the super-state must be present in all states of the super-state. In other words, output action in the observation projection represents option of the model to emit the action while input action represents obligation to accept it.

Fig. 6 contains a fragment of a provision-driven computation. There are states connected by transitions labeled by internal, input and output actions. The leftmost state is the initial one while the rightmost state is a final state. The rest of the LTS is represented by the dashed line. The corresponding observation projection (Fig. 7) consists of three super-states. The super-state k is the initial state, since one of the states it represents is the initial state of the provision-driven computation. Moreover, there is no transition labeled by an input action leaving k since not all of the states are ready to accept `?p`. On the other hand, there are two output transitions. The one labeled by `!q` leads to the super-state l, while the other one leads to the part of the observation projection which is not depicted. In contrast to k, there is an input transition leaving the super-state l, since all the states in l are ready to accept `?p`. The target states of these transitions are not distinguishable by the environment. Thus, they all belong to the super-state m.

The definitions provided in this section gradually simplify an open TBP model so that, in the end, the observation projection is just a transition system labeled by input and output actions—similarly to interface

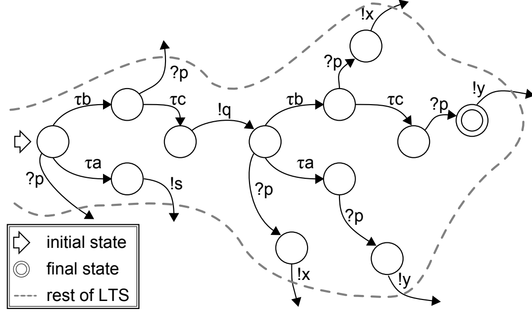


Fig. 6. Fragment of provision-driven computation

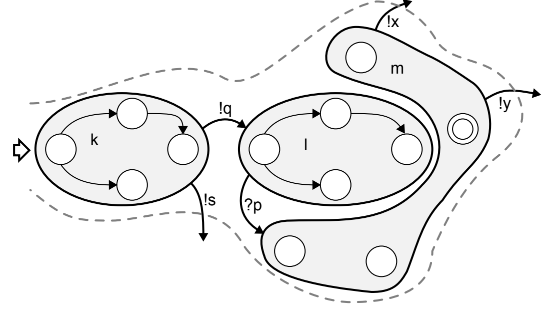


Fig. 7. Fragment of observation projection

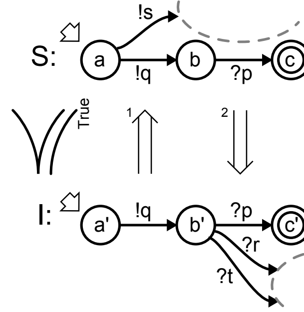


Fig. 8. Alternation simulation example

automata. There are, however, two significant differences. The transition system is deterministic (there are no internal actions and each state contains at most one outgoing transition for each external action). Second, there is also additional termination information associated with super-states.

Determining whether the observation projection $C^k_{OP}(I)$ refines $C^k_{OP}(S)$ is based on the parameterized alternation simulation.

Definition 31 (Parameterized alternation simulation). Let I and S be observation projections of TBP models. Let S_I be the set of states of I, S_S be the set of states of S, δ_I and δ_S be the transition functions of the respective observation projections. Moreover, let $P \subseteq S_I \times S_S$ be a relation. Then, we call the relation $\preceq_P \subseteq S_I \times S_S$ *parameterized alternation simulation* if

- $\forall (s_I, s_S) \in \preceq_P : (s_I, s_S) \in P$
- $\forall (s_I, s_S) \in \preceq_P : \delta_I(s_I, !m) = s'_I \Rightarrow \exists s'_S : \delta_S(s_S, !m) = s'_S \wedge s'_I \preceq_P s'_S$
- $\forall (s_I, s_S) \in \preceq_P : \delta_S(s_S, ?m) = s'_S \Rightarrow \exists s'_I : \delta_I(s_I, ?m) = s'_I \wedge s'_I \preceq_P s'_S$

The relation is extended to observation projections using initial states as follows.

We say that I refines S with respect to the property P ($I \preceq_P S$) iff there is a parameterized alternation simulation $\preceq_P$ such that $s^0_I \preceq_P s^0_S$ where s^0_I is the initial state of I and s^0_S is the initial state of S.

The parameterized alternation simulation stems from the alternation simulation introduced for interface automata in [13]. The main purpose of the alternation simulation is to relate states of the individual observation projections that correspond to each other from the observer's point of view. Let E be an environment enclosing S and the result of composition does not contain any error. Then, when $C^k_{OP}(S)$ exercised by the environment E is in the state s_i, then $C^k_{OP}(I)$ exercised by E must be in the state s_s such that $s_i \preceq_P s_s$.

The additional predicate P allows specifying an additional property that must hold to address a specific error. In particular, $I \preceq_{True} S$ preserves bad activity that occurs in communication among I (resp. S) and its enclosing environment E, however, it considers neither no activity nor bad activity caused by internal communication within I.

Fig. 8 contains an example of two observation projections I and S such that $I \preceq_{True} S$. In particular, $c' \preceq_{True} c$ holds trivially, since there are no leaving transitions and $(c', c) \in True$. The implication 2 and

$c' \preceq_{True} c$ implies $b' \preceq_{True} b$ and the implication 1 and $b' \preceq_{True} b$ implies $a' \preceq_{True} a$. Since the initial states $a' \preceq_{True} a$ then also $I' \preceq_{True} S$

Lemma 4 (Transitivity of $\preceq_P$). Let A, B and C be observation projections of TBP models, and P be a transitive relation over their sets of states S_A,S_B and S_C (i.e. $(s_A P s_B) \wedge (s_B P s_C) \Rightarrow (s_A P s_C)$). Let $A \preceq_P B$ and $B \preceq_P C$. Then $A \preceq_P C$.

7.3.3. Preserving Bad Activity

Theorem 1 (Refinement w.r.t. BA up to k threads). Let I_{OP} resp. S_{OP} be an observation projection of a TBP model I resp. S for k threads. Let E_I^{BA} resp. E_S^{BA} be sets of bad activity error states in the observation projections. Let $BA(a, b)$ be a relation over set of states of observation projections such that

$$BA(a, b) \Leftrightarrow (a \in E_I^{BA} \Rightarrow b \in E_S^{BA}).$$

Then $I_{OP} \preceq_{BA} S_{OP}$ implies that I refines S with respect to bad activity up to k threads.

7.3.4. Preserving No Activity

To define a refinement relation preserving the no activity error additional information is needed in the observation projection. Let A_{OP} be an observation projection. The predicate *running* holds for each state of the observation projection that has to emit an output action. The predicate *running* is defined as follows.

Definition 32 (Running). Let s_{op} be a super-state of the observation projection A_{OP} and S be a set of states in the provision-driven computation represented by s_{op}. Then

$$\begin{aligned} running(s_{op}) = \quad & s_{op} \notin !F \\ & \wedge \\ & \forall s \in S \quad \exists s_\tau, !m, s' : s_\tau \in \tau closure(s) \wedge \delta(s_\tau, !m) = s' \end{aligned}$$

The definition consists of two properties of the state. If the set of states of the provision-driven computation represented by the super-state contains a state representing termination of all active threads ($s_{op} \in !F$), the whole model can terminate on its own while not emitting an output action. The second property states that for each state of the set, there has to be a path consisting of internal actions leading to a state s_τ producing an output action.

Theorem 2 (Refinement w.r.t NA up to k threads). Let I_{OP} resp. S_{OP} be an observation projection of a TBP model I resp. S for k threads. Let E_I^{NA} resp. E_S^{NA} be sets of no activity error states in the observation projections. Let $NA(i, s)$ be a relation over set of states of observation projections such that.

$$\begin{aligned} NA(i, s) = \quad & BA(i, s) \wedge (i \in E_I^{NA} \Rightarrow s \in E_S^{NA}) \\ & \wedge \\ & s \in F \Rightarrow (i \in F \vee running(i)) \\ & \wedge \\ & running(s) \Rightarrow running(i) \end{aligned}$$

Then $I_{OP} \preceq_{NA} S_{OP}$ implies that I refines S with respect to no activity up to k threads.

8. Experiment—TBP model of CoCoME assignment

We tested the capabilities of TBP on a model of a supermarket information system involving row of cashdesks, each featuring a number of devices (keyboard, credit card reader, etc.), and a central database of goods in the store. The model is taken over from the CoCoME contest assignment [28]. The goal of the CoCoME contest was to compare strengths and weaknesses of different modeling approaches. In particular, each participant provided a model using their own formalism. The approaches of contest participants differ in both goals and means. Some approaches aim at performance modeling and prediction (Paladio, Klapper), while other aim at the functional correctness (rCOS, Java/A, CoIN). The means used by approaches aiming at functional correctness range from finite automata (CoIN), to those employing concepts of preconditions/postcontidion

formulas. When compared to goals and means of TBP, the most related approaches used in CoCoME were CoIN, Java/A, and our BP.

By providing a TBP model of the CoCoME example, we can compare it to the models provided by participants [31] as well as to the Java implementation, which was created as a part of the assignment. Unfortunately, there is no model of CoCoME in Interface Automata, the formalism which inspired definition of refinement in TBP. On the other hand, in contrast to TBP, we consider Interface Automata to be more a theoretical concept than a specification language intended for applications.

The TBP model of the CoCoME assignment is based on the architecture we already created for the EBP model [10]. When compared to the other specification languages, it was straight-forward to express multiple bindings in the architecture. In particular, neither reactions, nor provisions of a component providing a method to several other components (e.g., EnterpriseServer shared by many StoreServers) need to specify the particular degree of parallelism. Moreover, the sync keyword approved very useful for modeling mutual exclusion and it significantly simplified the specification of buses. All in all, the means provided by TBP enabled us to create a model that resembles the real implementation more closely than models in CoIN, Java/A and BP.

The TBP model of CoCoME as well as the tool for TBP checking (partially work in progress) is available at [7]. Even though the CoCoME assignment is far from being trivial, the initial version of the model was crafted roughly in one day. The analysis of the model revealed a deadlock (in bus access) and also a violation of provisions (the provision of StoreLogic did not allow one to process a sale while the StoreLogic component was processing accepting goods from another store). Specifically the latter would be very hard to discover manually.

9. Evaluation and Discussion

One of the objectives of the TBP design was to provide a formalism that would be simple enough for use by practitioners during day-to-day development. To achieve this goal, both TBP syntax and semantics are designed to be close to common imperative languages. At the same time, another goal was to be able to benefit from formal analyses of models specified upon an LTS background. An important requirement in the scenario was that verification of built-in properties and of refinement is decidable so that both could be analyzed by a tool.

In terms of the Goals, stated in Chapter 4, both of them have been achieved: (1) The syntax of the TBP formalism builds upon the constructs (if, switch, while, sync) and abstractions (threads, types, variables) present in imperative programming languages, thus being easier to comprehend and use by a developer in comparison to the traditional behavior modeling approaches, such as automata and process algebras. (2) The proposed TBP formalism supports reasoning about component composability and refinement, which provides a developer platform supporting both the bottom-up and top-down design. Here, specific kinds of errors, inherent to component applications, are detected (bad activity, no activity). Furthermore, thanks to fulfilling Goal 1, it is possible to reason about conformance of the code (implementation) with the corresponding behavior model (specification)—this, however, is beyond the scope of this paper.

Below we discuss other important TBP properties which make behavior modeling of components to be implemented in imperative object-oriented languages convenient.

Specification of Provisions. Provisions specify how an environment is expected to call the methods of the component. This information is not present in the code itself, at least not explicitly, therefore the developer should state the requirements of the component put on its environment within the TBP specification.

The language of provisions is inspired by BP [3]. In contrast to BP, method-call-related events are equipped with parameters and return values of enumeration types. This allows specifying assumptions on the environment with a finer granularity than just imposing particular sequencing of method calls.

Another important aspect is the granularity of assumptions' specification. For a component's model, several provisions guarding different sets of methods are allowed. Some methods can be omitted by the provisions section—these can be invoked arbitrarily by the environment, even in parallel. At the same time, a method can be guarded by several provisions and then the environment has to follow all of them. This approach supports separation of concerns.

	Models		TBP		Stuck	IA Error
	A	B	Bad Act.	No Act.		
a)	!a, !b	?a	Yes	No	No	Yes
b)	!b	?a	Yes	No	Yes	Yes
c)	?b	?a	No	Yes	Yes	No
d)		?a	No	No	Yes	No

Fig. 9. Communication errors in composition of A and B

Communication. The LTS specified by the model captures parallel interleaving of activities executed by individual threads. Individual threads influence each other only by modifications of state variables (this includes locks).

As long as the parallelism is not explicitly limited in provisions, a method can be invoked as many times in parallel as the environment requires.

In contrast, when process algebras are applied in component behavior modeling, every component is typically modeled by a single process. Communication among components is represented by the synchronization of actions representing method calls. A direct consequence is that method calls are represented as if communication among components was asynchronous—(a) the method call is not accepted if it is not explicitly awaited by the target process and (b) the caller process continues in the computation not waiting for a result. This is far from the semantics of method calls in imperative languages.

While the issue (b) can be resolved by using pairs of actions—an output action at the source process is immediately followed by the corresponding input action waiting for the result (and vice versa), the issue (a) requires further attention—it is most striking when the user wants to model a method which can be invoked in parallel as many times as the environment requires. As an aside, all Java classes exhibit such behavior by default. Process algebras (CSP in particular) support recursion which allows unlimited number of processes starting in parallel, as soon as required by the environment. This however, makes the formalism more complex and the state space of an open component system infinite.

Composition. In contrast to process algebras, composition of specifications in TBP is defined at the syntax level as union of the corresponding sets. The semantics defines how to create an LTS for an open model (observation projection). Since there is no means for composition of two observation projections, two open models are composed at the syntax level into a single model (possibly a closed one) which is transformed into an LTS.

Communication Errors. The definition of communication errors supporting TBP is based on comparison of the behavior actively performed by threads and description of the behavior passively expected by provisions. Since both of them can be expressed as a set of traces, the required properties can be stated as inclusion of these sets.

When comparing the actual behavior to the expected behavior at the trace level, there are hardly any errors other than bad activity and no activity. However, individual formalisms targeting (component) behavior modeling differ significantly in the way traces are constructed, specifically in dealing with non-determinism and performing internal/external choice. An important aspect related to non-determinism is the "degree of optimism" the formalism takes—whether a model containing an error state is considered erroneous: In case of the optimistic approach (Interface Automata), the model is considered correct if there exists an environment making the error state unreachable, while in the pessimistic approach existence of an error state implies the model is erroneous. In TBP, in particular, the error is reported if it can occur, i.e., the pessimistic approach is taken.

Even though it might look simple, there is a caveat to defining communication errors: Fig. 9 illustrates the differences among the TBP errors (bad activity and no activity, introduced in [2, 3]) and the stuck error introduced in [14] and discussed in [15]. Roughly, stuck error "combines" both bad activity and no activity. Fig. 9 presents pairs of label transition systems demonstrating differences among the errors. In each row, the state at the left-hand side is always an initial state while a double circle represents a final state.

The key differences are characterized by the following four state transitions: The pair (a) demonstrates the difference between the bad activity error and stuck error. While in TBP composition of A and B is considered to be erroneous because of the !b action from A is not accepted by a counterpart in B, the same situation does not cause a stuck error, since it suffices that at least an action is accepted by the counterpart (!a and ?a). Composition of the pair (b) produces a stuck error—no action leaving the initial states is accepted. This composition pair also reflects the difference between a stuck error and no activity. Since the definition of no activity, unlike of stuck error, considers final states and cannot occur at final states, there is a stuck error but not a no activity. The row (c) demonstrates that bad activity distinguishes input and output actions (? and !) while stuck error does not. Finally, the row (d) illustrates the difference between no activity and stuck error. While no activity can occur only in the states that are not final, stuck error does not consider final states at all.

The bad activity error has the same meaning as the error introduced in [13] for Interface Automata; nevertheless, there is no concept of no activity (deadlock) in [13].

Refinement. Fig 9 also demonstrates that the concept of refinement is based on the alternation simulation introduced in [13] for interface automata. For TBP, however, the relation is strengthened to preserve absence of no activity. The definition of refinement for TBP is provided in three steps.

First, the provision-driven computation transforms an open TBP model into an LTS similar to the interface automata—there are input, output, and internal actions. In the next step, observation projection is created in the form of a deterministic LTS. The final step checks the parametrized alternation simulation of two observation projections. This means to compare the options and obligations of the corresponding states from the "implementation" and "specification", i.e., the models being subject to refinement checking. First, the initial states are compared and then the process continues by comparing the pairs identified by the transitions labeled by the same actions.

By defining refinement in three steps, the theoretical framework behind gets more modular. In particular, the second step gathers the information from several states of a single provision driven computation indistinguishable by the environment into a single state of the observation projection. Most importantly, the second step resolves non-determinism; it determines what events important for preserving errors must or may occur. Thus, the third step does not consider these differences, since they are already abstracted away so that it can be defined in a straightforward way as a simple comparison of two states.

In this context, it is worth mentioning the special position of the refinement w.r.t. bad activity. The comparison of the sets of input and output actions in the third step ensures preservation of bad activity occuring on the component boundary on its own. At the same time, the comparison of the sets of actions forms a basis for relating pairs of states. The parameter formula is then applied on these pairs.

Expressiveness compared to Java. By comparing the Java implementation of the CoCoME assignment to its TBP specification, it becomes obvious that TBP specification omits technical details (such as accessing a database via specifically DerbyDB and using ActiveMQ), which make the implementation infeasible for code model checking (e.g., by JPF [17]). In principle, to employ such a code model checker for verification we would need a simplified Java model obtained by either (a) omitting details from the implementation, or (b) designing it from scratch. Such a model could be limited to the features available in TBP—limited number of threads, enumeration variables, and assertions in the code for expressing assumptions on the environment. Non-deterministic choices could be modeled by the Random class in Java. By JPF, such a Java model would allow checking deadlock freedom (no activity), as well as checking bad activity as obeying assumptions. To express the desired sequences of method calls, one would have to keep the related state (e.g., position in the sequence, state of the property automaton) manually in a Java variable and check its value in injected assertions or using a custom PropertyListener. In contrast, provisions available in TBP provide much more convenient way to express the desired sequences of method calls. Modeling the TBP reentrancy provision operator in Java would be also very tricky. Finally, even if we limited ourselves to enumeration variables, the size of the state space of the whole composition modeled in Java could become unfeasibly large for non-trivial

applications. In TBP, compositional verification using refinement in TBP implies partitioning of the state space, shifting thus feasibility limits further. No similar technique is available in JPF.

10. Conclusion

The proposed specification language, TBP, aims at narrowing the gap between imperative languages used in industrial software development and behavior specification languages.

Key achievements and benefits. While TBP remains a specification language, it tries to get as close as possible to the syntax and semantics of the mainstream imperative languages (Java in particular). Specifically, both the syntax and semantics are inspired by the imperative object-oriented programming languages the industrial developers are used to (Goal 1.a). Thus, the model can be crafted by similar means as the implementation—developers operate with the same concepts and the specification structure can be followed in the implementation as far as to the individual control flow statements. An important benefit is also that synchronization in TBP is inspired by the approach taken by Java. At the same time, TBP specification allows specifying assumptions on the environment, fulfilling thus Goal 1.b.

TBP allows analyzing models of applications in the context of component systems (Goal 2). In particular, TBP supports checking compositional correctness and refinement. The notion of correctness is based on verifying the actual behavior of a closed system against assumptions.

To sum up, TBP is a step towards narrowing the gap between a specification and implementation language. Strictly speaking, most of the TBP features and ideas have already appeared either in the world of behavioral modeling or in an implementation language. TBP, however, puts the ideas together in a novel way—in particular, unique is the option of reasoning on refinement of individual components' models, while the computation is driven by threads as in a real application.

When compared to previous formalisms from the Behavior Protocols family, TBP is a step forward in user friendliness and in the properties of formal framework.

The models related to an implementation are easier to write and understand in TBP than in other specification languages as demonstrated on the session manager example (Sect. 6) and its versions published in [31]. Regarding the formal framework, we consider beneficial that the refinement relation preserves both bad activity and no activity errors. Moreover, the refinement relation is modular, which helps enhance it in order to support additional errors. Even though there are other formalisms available supporting similar notion of refinement, to our best knowledge none of them follows the imperative object oriented languages as closely as TBP does. In particular, the behavior based on threads where each thread has its own stack allows the user to follow the object oriented design much closer than in any other formalism and still reason about refinement at the level of application architecture.

Future Work and Open Issues. There are still several features worth modeling, not supported in TBP. Specifically, these include: Dynamic thread creation, support for evolving architectures (software logic often involves dynamically created logical objects important for the high level behavior), and unlimited number of threads considered in the refinement.

We intend to address the last one as follows: To provide a refinement relation supporting unlimited number of threads triggered by environment, we want to add level of parallelism (based on the reentrancy operator) to the observation projection and define a relation that compares implementation and specification in a way that considers the actual level of parallelism. Nevertheless, dynamic thread creation and dynamic object support still remain open issues.

References

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component reliability extensions for Fractal component model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.

[2] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of the 2 nd International Workshop on Unanticipated Software Evolution*, 2003.

[3] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[4] A. Aldini, M. Bernardo, and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer, 2010.
[5] L. d. Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.
[6] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, CMU, 1997.
[7] Badger—Verification of component behavior specification. `http://d3s.mff.cuni.cz/~sery/badger`.
[8] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
[9] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.
[10] T. Bureš, M. Děcký, P. Hnětynka, J. Kofroň, P. Parízek, F. Plášil, T. Poch, O. Šerý, and P. Tůma. CoCoME in SOFA. In *The Common Component Modeling Example: Comparing Software Component Models*, pages 388–417, Berlin, Heidelberg, 2008. Springer-Verlag.
[11] I. Černá, P. Vařeková, and B. Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06)*, volume 182 of *ENTCS*, pages 39–55. Elsevier Science Publishers, June 2007.
[12] E. M. Clarke, N. Sharygina, and N. Sinha. Program compatibility approaches. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Lecture Notes in Computer Science*, volume 4111, pages 243–258. Springer, Springer, 2005.
[13] L. de Alfaro and T. A. Henzinger. Interface Automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.
[14] C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-Free Conformance. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2004.
[15] C. Fournet, T. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance theory for ccs. Technical report, Microsoft Research, July 2004.
[16] R. Grimes and D. R. Grimes. *Professional Dcom Programming*. Wrox Press Ltd., Birmingham, UK, 1997.
[17] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International (UK) Ltd., 1985.
[19] J. Kofron. Checking software component behavior using Behavior Protocols and Spin. In *Proceedings of Applied Computing 2007*, pages 1513–1517, Seoul, Korea, March 2007.
[20] K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O Automata for Interface and Product Line Theories. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.
[21] G. T. Leavens and M. Sitaraman, editors. *Foundations of component-based systems*. Cambridge University Press, New York, NY, USA, 2000.
[22] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference, ESEC '95 , Barcelona*, 1995.
[23] V. Matena, B. Stearns, and L. Demichiel. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. Pearson Education, 2003.
[24] R. Milner. *Communication and Concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.
[25] OMG Group. CORBA Component Model Specification. Technical report, OMG Group, 2006.
[26] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.
[27] T. Poch. *Towards Thread Aware Component Behavior Specifications*. PhD thesis, Charles University, Prague, 2010.
[28] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.
[29] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
[30] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.
[31] Modelling Contest: Common Component Modelling Example, `http://agrausch.informatik.uni-kl.de/CoCoME`.