Department of Distributed and Dependable Systems
Technical report no. 2011/05
September 13, 2011

Refinement between TBP and Java Implementation of Components

Jan Kofroň, Pavel Jančík, Pavel Parízek

This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the Grant Agency of the Czech Republic project P202/11/0312.

Abstract. An important correctness property of software built in a modular way is behavior specification of its particular components. Only then one can reason about communication correctness and properties of particular components. Since it is much more effective to do so at the level of behavior models, establishing a correspondence between behavior specification and implementation becomes an important part. In this paper, we present BeJC—a tool for consistency checking between software component implementation and its behavior specification. We also discuss practical experience with the tool and its application in the software development process.

1 Introduction

An important correctness property of software systems built in a modular way, i.e., from well-defined components, is that each component behaves only according to a given specification. This can involve functional (degree of parallelism, invoked methods) as well as extra-functional (e.g., performance, reliability) properties. As to the functional properties, the more precise behavior specification is available, the more one can say about the component without inspecting its implementation. A relatively precise way is specification of allowed sequences of method calls on other components in the system the component performs. We call the property of error-free communication among components *the component interaction compatibility*. The valid sequences of method calls can be specified in a number of way, e.g., in the form of transition systems, e.g., LTS, finite automata, and in a higher-level language, e.g., LOTOS [14], CSP [7], Threaded Behavior Protocols (TBP) [10]. The property of component interaction compatibility holds for a given system if:

- the implementation of each component is *consistent* with its behavior specification, i.e. each component performs only the method calls and in such order that is allowed by its specification, and
- there are no interaction errors between components of the system, such as performing a method call unexpected by the target component.

While many techniques for detecting interaction errors at the level of behavior specifications were proposed in the past [1, 5], much less work has been done on checking consistency between the component implementation (e.g., in Java and C) and its behavior specification. Besides our previous work [11], we are aware only of one existing approach for Java [6], which is, however, only partially automated.

In this paper, we present the BeJC tool for automatic checking of consistency between the Java implementation of a software component and its behavior specification in the TBP language. It implements the consistency checking algorithm proposed in our previous work [11] with modifications needed for the new TBP specification language. The BeJC tool is publicly available for download at the web site `http://d3s.mff.cuni.cz/projects/formal_methods/bejc`.

2 Threaded Behavior Protocols

A behavior specification in TBP consists of five parts: *types*, *variables*, *provisions*, *reactions*, and *threads*. In the type section, custom enumeration types are defined. These are used as types of local variables defined in the variable section. These variables are used for storing information across particular method calls. *Provisions* define permitted usage of the component, i.e., the sequences of method calls that the component expects from other components—its environment. The component must handle these sequences, which means it should not end up in a deadlock state or behave in a way different from what is specified in the *reactions* and *threads* sections. *Provisions* can be seen as an assumption the component takes about its environment. Syntax of the *provisions* section is similar to regular expressions over terms of the form ?itf.m that expresses acceptance of a method call on a provided component interface (itf stands for an interface name and m stands for a method name). It supports the standard regular operators (;, +, *) and the parallel operator | that permits any interleaving of method call sequences defined by its operands. *Reactions* define how the component reacts to a particular method call, in terms of invoking methods on its required interfaces and modifying the content of its local variables. Syntax of the *reactions* is inspired by Java-like programming languages. The basic statement is a method call (!itf.m(param)). More complex statements can be constructed using

the sequencing operator ; and control structures well-known from programming languages (e.g., while and if-then-else). The ? character used at the place of a condition, e.g., in the if statement, represents a non-deterministic choice. The *threads* section specifies permitted behavior of threads internally created by the component. Syntax of the *threads* section follows the syntax of the *reactions*. Semantics of the TBP language is defined using LTS. In particular, correct communication of particular components on the level of TBP and the refinement relation are not based on a simple trace set inclusion, but it is based on *alternation simulation* defined in [2]. More details on TBP can be found in [10].

```
component IpAddressManager {
  provisions {
    ?IDhcpServer.Start ; (
      ?IDhcpServer.RequestAddr +
      ?IDhcpServer.ReleaseAddr
    )*
  }
  reactions {
    IDhcpServer.RequestAddr {
      !IIpMacDb.GetAddress ;
      if (?) {
        !IIpMacDb.Add ;
        !ITimer.SetTimeout
      }
    }
  }
}
```

```
class IpAddressManagerImpl
      implements IDhcpServer
  private IIpMacDb db;
  private ITimer timer;

  String RequestAddr(byte[] mac) {
    String ip = db.GetAddress(mac);
    if (ip == null) ip = allocIP();

    Date expTime = new Date(...);
    db.Add(mac, ip, expTime);
    //timer.SetTimeout(expTime);

    return ip;
  }
}
```

Figure 1: Example: TBP specification and Java implementation

3 Compliance

Figure 1 shows a fragment of the TBP specification for the IpAddressManager component. The component manages IP addresses for clients on the network. The *provisions* section states that Start must be invoked first on the component through its IDhcpServer interface; then it is possible to call RequestAddr or ReleaseAddr repeatedly. The *reactions* section specifies that, upon accepting a call of RequestAddr, the component must invoke GetAddress and then optionally Add and SetTimeout in this order. The example contains an inconsistency between the TBP specification and the Java implementation listed at the right-hand side of Figure 1—the SetTimeout method is not invoked after Add in response to the RequestAddr call. This is the kind of errors that BeJC can detect.

To be more precise as to the relation between the specification and code, the implementation of the component under verification (1) has to accept any calls on its provided interfaces in the order that is allowed by *provisions* and (2) is allowed to invoke just those methods on its required interfaces and in the order that are specified in the *reactions*. Since the implementation does not feature any explicit *provisions*, i.e., the provisions are empty and do not restrict the usage of the component, any sequence of methods can be called on the component provided interfaces. The aforementioned method verifies the existence of the alternation simulation between TBP and the Java code and if no error is discovered, we know that the implementation refines the TBP specification; in other words, the given TBP specification is an abstraction of the implementation. This way, absence of communication errors of communicating components captured at the level of TBP is after the code compliance verification assured also on the

implementation level.

The aforementioned claims are justified by the following algorithm and theorems. We will take advantage of the formal framework defined in [12]. The following *checking algorithm* describes the way BeJC works.

1. The component environment from the provisions is generated: For each parallel branch of the *combined provisions for k threads* [12] a thread in the environment code is created. The code consists of the sequences of provided methods' calls that are specified by the parallel branch. The parameter values for these method calls are taken from a user-defined database.

2. The code of the environment and the component is compiled and the compliance checking is run.

3. The tools repeats the following step until all options with respect to thread scheduling and parameter values are tried: A thread (either of the environment or of the component) is selected and its step is performed. If the step is a method call on a component's required interface, the BeJC tool checks whether this call is, with respect to the history of already performed actions, allowed by the TBP specification.

4. If there is no disallowed method call on a required interface (according to the TBP specification), no deadlock, and no uncaught exception (checked by JPF) found during the checking process of the previous step, the algorithm returns *true*, otherwise it returns *false* along with the error trace (a sequence of calls leading to this state).

Definition 1. *Let C_I be an implementation of a software component C and S be a TBP specification which does not contain the general re-entrancy operator. We say that S is a* TBP specification induced by C_I *if:*

(i) The provisions are empty, i.e., no restriction on the order of calls of the component provided interfaces is stated.

(ii) For each method on a provided interface of C the set of all possible traces of method calls on the component required interfaces performed by C are a subset of the set of the traces specified by the corresponding reaction of S, if necessary also with respect to the history of calls.

(iii) If there are internal threads the component creates, the subset relation holds in the same way for the set of threads' traces of C and the threads section of S, otherwise the threads section is empty.

An induced TBP specification is an abstraction of the implementation in the sense of observable behavior. It is necessary to take the history of calls into account, since the component can remember information (its state) in a variable and react in a different way in various states.

Theorem 2. *Let S be a TBP specification and I be a TBP specification induced by a component C. If the checking algorithm returns* true, *then I refines S.*

Proof. For a contradiction, suppose that for a TBP specification S and a component implementation C the checking algorithm returns *true*, but I does not refine S.

In order that I refines S, there must be an alternation simulation between initial states of observation projections P_S and P_I of S and I, respectively. Let $s_S \in P_S$ and $s_I \in P_I$. There are three conditions that must be satisfied:

1. $\forall (s_I, s_S) \in \preceq_E$: $(s_I, s_S) \in E$

2. $\forall (s_I, s_S) \in \preceq_E$: $\delta_S(s_S, ?m) = s'_S \Rightarrow \exists s'_I : \delta_I(s_I, ?m) = s'_I \wedge s'_I \preceq_E s'_S$

3. $\forall (s_I, s_S) \in \preceq_E$: $\delta_I(s_I, !m) = s'_I \Rightarrow \exists s'_S : \delta_S(s_S, !m) = s'_S \wedge s'_I \preceq_E s'_S$

Here, E is a relation that relates the (error) states of P_I and P_S, which is equal to true in this case, since there is no a-priori relation between the (error) states of I and S (and hence between P_I and P_S). Next, the provisions of P_I are empty, in other words, there is no restriction on the calls on provided interfaces. The second condition is thus trivially satisfied as well. The only condition that can be violated (and hence disallow the refinement between I and S) is the third one. To violate the third condition, there has to be an sequence of states $t = i_0 i_1 \ldots i_n$ in P_I and a sequence of states $s = s_0 s_1 .. s_n$ in S_I such that $\forall k = 0..n-1 : i_k \preceq_E s_k$. Next, let there be a transition $!m$ from i_n such that there is no such transition from s_n. Following the construction of the observation projection [12] we observe that the state s_n was constructed from a set of states such that none of them contained the transition $!m$ (otherwise s_n would contain such a transition as well). Moreover, again from the definition of observation projection we know that there is no other trace from the initial state of P_S to another state of P_S that would share the same sequence of labels—the state s_n is unique in this sense. The sequence t is also present in I and hence in C. Since the *checking algorithm* uses the provision of S to construct the environment of C, the sequence t is also executed by the algorithm. After reaching the state i_n, it return *false* since there is no $!m$ transition from the corresponding state s_n. This is a contradiction to our assumption made at the beginning of the proof and the theorem is proven. □

Theorem 3. *If the* checking algorithm *returns true for all components of a closed system and there is no communication error in the component composition at the TBP level, then there is no deadlock, no uncaught exception, and the method calls on provided interfaces of each component in the system comply with the corresponding provisions.*

Proof. Theorem 3 is a direct corollary of Theorem 2 and the correctness of the composition verification algorithm for TBP defined in [12]. □

Still, there is an issue regarding construction of the component environment that is worth mentioning at this place. As aforementioned, there is a "parameter database" that defines the values for the provided methods called by the environment on the component. So far, the database construction is up to the user of BeJC. Since the tool works in the explicit model checking mode, there is no way to practically try all possible values for a given type, e.g., strings, integers, and classes. It is up to the user to include all the values that make the tool cover all the possible situations in the component under verification. Only then Theorem 2 and Theorem 3 hold.

4 Tool Description

The BeJC tool checks consistency between Java code and TBP specification using state space traversal. It consists of four modules: TBP library, environment generator, TBP checker, and Java PathFinder (JPF) [9]. The architecture of the tool and flow of information among its modules are shown in Figure 2.

The input of the tool is the Java implementation, description of the component (metadata), and the TBP specification. TBP library parses the TBP specification and creates two intermediate representations: abstract syntax tree (AST) and a state transition system. The environment generator creates an abstract environment for the component from the metadata and the AST representation of provisions in the TBP specification. The abstract environment is a nondeterministic program that performs all method call sequences allowed by provisions on the component and calls each method with different combinations of parameter values—possible values must be provided by the user as a part of the metadata. The Java program composed of the abstract environment and Java implementation of the component is the input for actual checking.

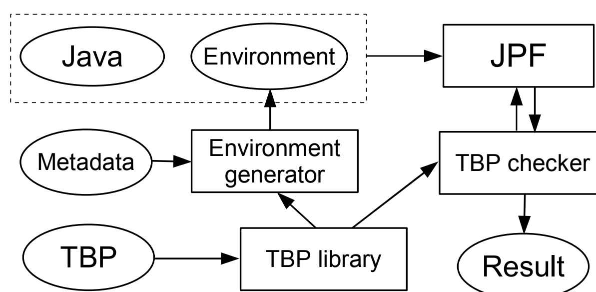

Figure 2: Architecture of the BeJC tool

The process of checking involves parallel traversal of (i) the state space of the Java program and (ii) the state transition system derived from the TBP specification. JPF traverses the Java program's state space and notifies the TBP checker about method call events (invocations and returns) as they occur during program's execution. TBP checker, implemented as a plug-in for JPF, checks for each event whether it violates the TBP specification—this happens if there is no matching transition from the current state of the transition system. This way, JPF with the TBP checker verify that the Java implementation responds to all permitted calls on the component's interfaces in a way allowed by the reactions and threads sections of the TBP specification.

The output of the BeJC tool is the result of checking and a counterexample in the case of a negative answer. The counterexample has two parts: execution path in the Java program and a current path in the transition system.

Additional details about BeJC, such as description of the checking algorithm and the approach used to generate the abstract environment, can be found in [8].

References

[1] J. Adamek and F. Plasil. Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance, 17(5), 2005.

[2] L. de Alfaro and T. Henzinger. Interface automata. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9). 2001. ACM, New York, NY, USA, 109-120.

[3] L. Bulej, T. Bures, T. Coupaye, M. Decky, P. Jezek, P. Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. CoCoME in Fractal, LNCS, vol. 5153, 2008.

[4] The CLIF Project, `http://clif.ow2.org/`

[5] D. Giannakopoulou, J. Kramer and S.-C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach, Autom. Softw. Eng., 6(1), 1999.

[6] D. Giannakopoulou, C.S. Pasareanu and J.M. Cobleigh, Assume-Guarantee Verification of Source Code with Design-Level Assumptions, In ICSE 2004, IEEE.

[7] C. A. R. Hoare: Communicating Sequential Processes, Prentice-Hall, 1985, ISBN: 0-13-153271-5

[8] P. Jancik. Checking Compliance of Java Implementation with TBP Specification, Master Thesis, Charles University, 2010.

[9] Java PathFinder, `http://babelfish.arc.nasa.gov/trac/jpf/`

[10] J. Kofron, T. Poch, and O. Sery. TBP: Code-Oriented Component Behavior Specification, In SEW-32, IEEE, 2009.

[11] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, In SEW-30.

[12] T. Poch, O. Šerý, F. Plášil, J. Kofroň. Threaded Behavior Protocols, accepted for publication in Formal Aspects of Computing, LNCS, 2011

[13] The Q-ImPrESS project, `http://www.q-impress.eu`

[14] T. Bolognesi and E. Brinksma: Introduction to the ISO specification language LOTOS, Comput. Netw. ISDN Syst., 14/1, Elsevier Science Publishers B. V., 1987