BeJC: Checking Compliance between Java Implementation and Behavior Specification[*]

Pavel Jančík[1] jancik@d3s.mff.cuni.cz
Pavel Parízek[1,2] parizek@d3s.mff.cuni.cz
Jan Kofroň[1] kofron@d3s.mff.cuni.cz

[1] Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic
[2] School of Computer Science, University of Waterloo, Canada

ABSTRACT

An important correctness aspect 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 a method for verifying *compliance* between behavior specification of a software component and its Java implementation. We also discuss practical experience with the BeJC tool that implements the compliance checking algorithm 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 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. Treating a particular component as a black box, a relatively precise way is specification of allowed sequences of method calls on other components in the system the component performs as well as the sequences it is able to accept. We call the property of error-free communication among components *communication correctness.* The valid sequences of method calls can be specified in a number of ways—in the form of transition systems, e.g., LTS, interface automata [1], or in a higher-level language, e.g., LOTOS [2], CSP [3], Threaded Behavior Protocols (TBP) [16].

[*]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.

It is a hard task to assure communication correctness on an implementation level. However, using a suitable formalism (such as TBP), it is possible to divide the task into two simpler questions (see Fig. 1). The property of communication correctness (on an implementation level) holds for a given system. if:

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

While many techniques for detecting composability, i.e., communication errors at the level of behavior specifications, were proposed in the past [4, 11], much less attention has been paid on checking compliance between the component implementation (e.g., in Java and C) and its behavior specification. Besides our previous work [17], we are aware only of one existing approach for Java [12], which is, however, only partially automated.

In this paper, we present an algorithm for checking compliance between a Java implementation of a software component and its behavior specification in the TBP language. It implements the compliance checking algorithm proposed in [17] with extensions needed for the TBP specification language [5].

The rest of the paper is organized as follows. Threaded Behavior Protocols are briefly introduced in Section 2. In Section 3, the relation between implementation and specifi-

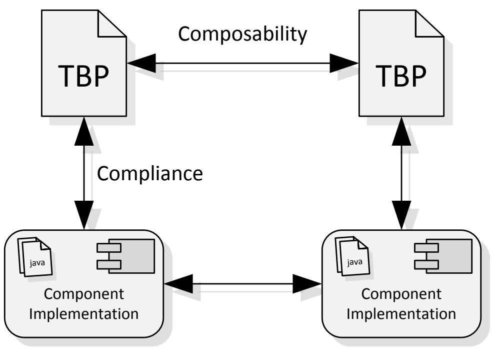

Figure 1: Communication correctness

cation is described. Section 4 discusses the implementation of our approach in the BeJC tool. Evaluation of our tool can be found in Section 5. In Section 6, we compare our tool to other similar approaches, while Section 7 proposes directions of future work and concludes the paper.

2. THREADED BEHAVIOR PROTOCOLS

Threaded Behavior Protocols (TBP) are a formalism for specification of software components' behavior in hierarchical component systems (e.g., in SOFA [24] and Fractal [25]). The idea is, at design time, to create behavior specification of each component in the system under development and verify (1) the composability of particular components and (2) the refinement across the adjacent nesting levels. Having these properties verified at the model level, one can ask about the correspondence between implementation of primitive (i.e., those on the lowest nesting level) components and its specification in TBP, thus making the reasoning about the entire system behavior complete.

TBP describes how single component (primitive or composed) interacts with the rest of the system. TBP is mainly focused on specification of the observable behavior (that is method calls accepted on provided and emitted on required interfaces of components).

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 (component) local variables defined in the *vars* section following definition of types. Variables are used for storing information across particular method calls. There are no pre-defined types [1].

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. *Provisions* can be seen as an assumption the component makes about its environment. On the other hand, if the environment behaves just in terms of the component's *provisions*, the component guarantees to behave in a way that is specified in the rest of the TBP (the *reactions* and *threads* sections) specification.

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.

Finally, the *threads* section specifies permitted behavior of threads that are created by the component itself. Behavior of *threads* consists of, similarly to *reactions*, invoking methods on required interfaces of the component and modifying content of component's variables. All the threads are supposed to start at the beginning, no dynamic (run-time) creation of threads is allowed.

Let us now provide a more formal look at TBP. First, we describe the syntax and semantics, which will be followed by description of relations between particular TBP specifications that are subjects to verification.

```
component CardReader {
  types {
    states = {READER_ENABLED, READER_DISABLED}
  }
  vars {  states state = READER_ENABLED }

  provisions {
    ?CardReaderCntl.expressModeDisabled()*
    |
    ?CardReaderCntl.expressModeEnabled()*
  }

  reactions {
    CardReaderCntl.expressModeDisabled() {
      state <- READER_DISABLED;
      !LightDisplay.displayExpress()
    }
    CardReaderCntl.expressModeEnabled() {
      state <- READER_ENABLED;
      !LightDisplay.displayNormal()
    }
  }

  threads {
    T1 {
      while (?) {
        if (state == READER_ENABLED) {
          !EventDispatcher.creditCardScanned();
          !EventDispatcher.PINEntered()
        }
      }
    }
  }
}
```

Figure 2: Example: TBP specification

2.1 Provisions

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 (sequence ';', alternative '+', repetition '*') and the parallel operator '|' that permits any interleaving of method call sequences defined by its operands.

Typically the '|' operator is used to describe the fact that multiple components uses the component being specified in parallel. On the source code level all possible interleaving of method bodies are checked.

As to the semantics of *provisions*, the expressions define a regular language, i.e., a set of words (traces) over the events. As an aside, once the expression contains the repetition operator *, the set of traces is infinite.

2.2 Reactions

Reactions specify how the component reacts to the methods calls accepted on its provided interfaces. Syntax of this section differs from the one of *provisions*, it resembles imperative programming languages, such as Java. For each method of each provided interfaces, a (possibly empty) reaction is specified in the form:

itf.m { *body* }

The body of the method reactions can contain following types of actions: An empty action NULL, method call !itf.m(par1, ...) (The informal meaning is invocation of a method on a required interface.), variable assignment var <- val, return from the method, conditional branch if and while cycles. In conditions the ? character means nondeterministics choice, the implementation is free to choose any branch.

[1]except for the mutex type (not described here).

The semantics is defined by LTSA—Labeled Transition System with Assignments. It is basically LTS enriched with variables, guards, and assignments. More details about the formalism of LTSA itself is beyond the scope of this paper and can be found in [18].

2.3 Threads

The *threads* specify behavior of the component threads, if there are any. Both syntax and semantics stem from those of *reactions*, except for the method name at the beginning, of course, that is replaced by a thread name.

Example.

An example of the TBP specification can be found in Fig.2. It specifies behavior of a credit card reader at a supermarket cash desk. The card reader is always either enabled (represented by the `READER_ENABLED` value of the `state` variable) or disabled (`READER_DISABLED`). In any case, the component is accepting calls on its `CardReaderCntl` interface switching between the two modes. Upon each such a call, it changes the value of the `state` variable and calls the `Display` component via its `LightDisplay` interface to display information about the current mode. In the normal mode (`READER_ENABLED`), the internal thread of the component, named `T1`, issues calls on the `EventDispatcher` interface about the events related to scanning a credit card and entering PIN. Since we do not capture any user behavior, the calls corresponding to particular events are issued randomly in a cycle, guarded by the condition upon the `state` variable value.

2.4 Composability and Refinement

Once having behavior of all components in a system specified in TBP, two correctness relations can be verified: composability and refinement. Composability means correctness of composition, in other words a set of components on a particular level of nesting is composable if there, according to their TBP specification, no error in their mutual communication can appear. An error usually arises as a consequence of violation of the provisions or in the form of deadlock. Violation of the component's provisions means that methods are called on interfaces of a component in a sequence that is not in the set specified by the provisions.

Refinement expresses the ability of a component B to replace a component A. After such replacement, the component B is required to handle all sequences of method calls upon its provided interfaces that were handled by the original component. The refinement relation is useful for verification of the refinement relation between a composed component and its subcomponents. Since a composed component usually does not contain any implementation per se, but just by means of its subcomponents, its behavior specification is the only behavior model available. Verification of the refinement relation down the nesting hierarchy enables one to reason about composability on higher levels of nesting, thus viewing the system on a more abstract level. More details on composability and refinement are beyond the scope of this paper and can be found in [18].

Moreover, although the implementation does not feature any provisions specification, refinement between implementation of a primitive component and its TBP specification allows one to reason about composability of primitive components as well, using their TBP specifications. If there is no error in communication inside a set of components on the TBP level, the refinement relation guarantees absence of communication errors also on the implementation level .

This case of refinement is called *compliance* and is described in the following section.

3. COMPLIANCE

Fig. 3 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 Fig. 3—the SetTimeout method is not invoked after Add in response to the RequestAddr call.

To be more precise as to the relation between the specification and code, the implementation of the component under verification (i) has to accept any calls on its provided interfaces in the order that is specified by *provisions* and (ii) has to invoke exactly those methods on its required interfaces and in the order that are specified in the *reactions*.

The (i) condition holds trivially, since the implementation does not feature any explicit *provisions*, i.e, any sequence of methods calls on the component provided interfaces are accepted.

To verify the (ii) condition, first an artificial environment for the component implementation is created; the environment simulates all permitted usage of the component by the environment, i.e., calls all the provided methods of the component in the order that is specified by provisions in TBP.

As for method parameters, the user has to specify sets of possible values for each data type in these method calls. The environment then calls the component with all combinations of the values. The code of the component together with its environment is then taken as the input of verification; each time the component calls a method on one of its required interfaces (if any), the call is absorbed by the environment in a dummy way (by an empty method), returned, and it is checked that the call is at this point allowed by the reaction of the method in TBP. Each reaction specifies a set of traces (words over required method calls), and the implementation is required, upon each method call, to invoke a sequence of the set. If this holds for any combination of arguments and any thread scheduling, the implementation complies to the TBP specification. This way, error freedom of component composition at the TBP level is assured also on the implementation level. For more details on the refinement and for related proofs, we kindly refer the reader to [15].

4. IMPLEMENTATION

The compliance checking algorithm between Java code and TBP specification is implemented in the BeJC tool. It consists of three modules: environment generator, TBP checker, and Java PathFinder (JPF) [14]. The architecture of the tool and the flow of information among its modules are shown in Fig. 4.

The input of the tool is the Java implementation, de-

```
component IpAddressManager {
  types { }
  vars { }
  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) {
      // Mac does not have assigned pernament IP address
      // allocate dynamic IP from pool
      ip = allocIP();
    }

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

    return ip;
  }
}
```

Figure 3: Example: TBP specification and Java implementation

	LOCs	Running time	Memory	States	States per sec
CLiF – BladeInsertAdapter	512	321 sec	502 MB	182,773	569
CoCoME – CashDeskApp	305	1,106 sec	306 MB	1,625,413	1470
CRE Demo – IP Address Manager	173	17,344 sec	1,197 MB	11,084,101	639
Q-ImPrESS – Pricing Manager	215	4 sec	138 MB	2,613	653

Table 1: Empirical results

scription of the component (metadata) including a value database, and the TBP specification.

First, the environment generator creates an abstract environment for the component from provisions in the TBP specification. The abstract environment is a non-deterministic program that performs method calls in the sequences specified by provisions on the component provided interfaces. Parameters of the called methods are taken from the value database; all combinations of parameter values are checked. The Java program composed of the abstract environment, values database, and the Java implementation of the component form the input for actual checking.

The process of checking involves parallel traversal of (i) the state space of the Java program (the environment with the component) by JPF and (ii) the state transition system derived from the TBP specification by TBP Checker. JPF traverses the Java program's state space and notifies the TBP checker about events (invocations of and returns from the methods on component interfaces, thread creation and termination) as they occur during program execution. The TBP checker, implemented as a plug-in for JPF, checks for each event if it violates the TBP specification—it traverses the state space of the *provision driven computation* [18] (which is derived from TBP specification) and checks if a transition that matches the event exists.

During the checking JPF uses the depth-first search while traversing the state space, while TBP checker maintains the set of all reachable TBP states by events on the processed trace.

If the represented set of the TBP states become empty, the processed event is not allowed by the TBP specification, so implementation violates the specification.

We also extended the JPF state matching algorithm to take the TBP states into account. The program/JPF state does not contain previous events—different program traces can lead to the same program/JPF state. However, reachable TBP states depend on previous events, it means that for a single program/JPF state may exist multiple TBP protocol states, with different permitted behavior in the future. Therefore JPF used in the BeJC considers the same program states with different TBP states as distinct/different.

If a protocol violation is found, JPF provides a (re-)executable trace which leads to the error.

Quality of the value database is crucial, only if values permits execute all execution paths in the component, the tool can find all errors in the implementation. Moreover, the environment has to perform all the permitted sequences of events on the provided interfaces. In the default settings,

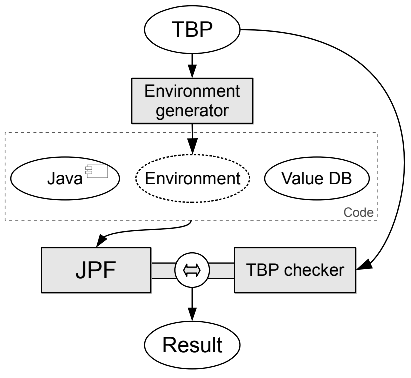

Figure 4: Architecture of the BeJC tool

repetition operators in the provisions are unrolled in the environment (and only a few initial iterations of loops are checked). The environment generator can be set up to perform unbounded number of loop iterations Such an environment typically generates more states.

Additional details about BeJC can be found in [13].

5. EVALUATION

Table 1 shows the results of application of BeJC to four components selected from four simple yet non-trivial examples introduced below. The results include lines of code of the tested component, the running time, memory consumption, the number of states, and the number of states processed per second by the checker.

CLIF [8] is a stress testing (benchmarking) framework implemented in the Fractal component model. The tested component (Blade Insert Adapter) is one of the largest core CLIF components. It asynchronously processes requests on the state changes of the blades (test probes and load generators). To be able to check the given component, we have to create a TBP protocol which models the usage of the component in a reasonable way and describe the behavior of the implementation and communication between provided interfaces and the internal thread. This application is focused mainly on testing the scalability of the BeJC tool—if a complex protocol and multiple threads are used.

CoCoME—Common Component Modeling Example contest [6] was aimed at comparison of various component models applied on the same case study—a supply chain and order management system of a typical supermarket. The selected component (CashDeskApplication) models a standard cash desk with a bar code scanner, card reader, keyboard, etc. We used its Fractal implementation [7]. The TBP protocol models the internal state of the component as well as its (limited) parallel usage.

CRE—Component Reliability Extensions [9] was a project that extended the Fractal component model with the support for Behavior Protocols. We have taken the CRE case study—a system for providing WiFi internet access at airports. We have checked the IPAddressManager component, which implements the internal logic of a DHCP server. The component has to accept requests from clients, timer callbacks (that terminates the IP leased period) and requests for configurations changes in a parallel way (three in parallel used interfaces), which implies quite a large state space.

Q-ImPrESS [19] was an EU project focused on cost-effective development, modeling, and evolution of service-oriented software. The picked Pricing simulator component is a part of the "eSOA Showcase". It is an example of typical business-logic code; it computes the price of orders, applies the discounts according to the volume, country, and customer type, etc. The component provides a single interface which is used synchronously, thus the number of the states as well as the checking time are relatively low.

None of the tests from Table 1 found an error, so the tests presents exploration of the whole state spaces. In our experience, the error (if present) is detected much earlier than the whole state space is traversed. When checking the IPAddressManager component, an inconsistency in its initial version was found in 6 minutes, while the error from Fig. 3 was detected in a second. In such a case the BeJC tool prints out a counter-example—an execution path in the Java program and the corresponding path in the TBP transition system (in terms of executed provided/required interface method calls/returns and threads which executed particular events).

The purpose of the evaluation was to focus on the scalability of BeJC, that is why only a single component from each project has been chosen. Later on, we plan to apply BeJC (along with the tool to check communication correctness on the TBP level) on a larger case study of a real-world component-based application to evaluate also application of the tool in the development process.

For the verified components, the checking process finished in a reasonable time. Nevertheless, state explosion would be an issue for large components and in the case when a TBP specification allows concurrent method calls on the component provided interfaces by several threads (a complex provision section in the TBP).

6. RELATED WORK

There are a number of approaches to behavior verification of software [2, 20, 21, 10, 22]. In this section, we only focus on those that work directly with the code, not just with its behavior model.

As to the tools for verification of Java code, we mention the Extended Static Checker for Java (ESC/Java2) [10]. It uses the JML annotations of Java source code and via the means of static analysis, it tries to discover common runtime errors in case of usage permitted by JML specification; it proves things locally, with respect to classes, methods and functions. Even though it is possible to express high level specification in JML, the overall consistency of the specification is to be assured by the user. Our approach is to automate the entire process as much as possible, i.e., to include both compliance checking described in this paper together with verification of composability at the level of the model (TBP).

A very related to our work is, of course, the Java PathFinder [14] itself. The tool consists of a custom Java virtual machine that explores the entire state space generated by a Java program with respect to different threads' interleaving and "non-deterministic" (random) values. A significant difference is that JPF requires a complete program (especially featuring a *Main* method); then it is able to discover problems such as uncaught exceptions, deadlock, null dereferences, and some types of data races.

In an empirical study [23], the authors focus on Object Protocols (OP) in the Java source codes. OP are similar to the TBP provision in the sense they describe the "permitted" sequences of method calls. In contrast to TBP which are used to describe complex component interactions, OP discovered in the code by the authors are quite simple. Only seven categories were used to describe most of the protocols. According to our experiences, components often exhibit behavior which does not match these categories. It is partially due to complex internal states of the components and the behavior derived from precise specifications (use cases). Note that in terms of OP, TBP protocols define both definition as well as usage (the provision section in TBP defines OP, while the reaction/thread sections describe usage of other components).

The last work we shortly mention here as related are Interface automata [1]. Although the specification language is not used for checking its correspondence to code, it was an inspiration for our refinement definition based on alternation simulation; by differentiating between provided and required method calls the alternation simulation provides a transitive relation fitting our needs.

7. FUTURE WORK AND CONCLUSION

From the perspective of the software development process, the BeJC tool can be applied during the testing and maintenance phase. BeJC also fits into the integration test well, where it can discover communication errors, incompatibilities and data races unforeseen during design and implementation of particular components.

In the maintenance phase, BeJC can be used especially for checking consistency after the implementation or specification of a component is modified, e.g., due to refactoring and emergence of new requirements. The tool has been integrated into the Q-ImPrESS development environment [19].

The need of the value database can be removed if symbolic execution approach is applied (symbolic JPF). We have not used this approach so far due to its scalability issues; however, we plan to try it in the future, as the required modifications of BeJC are not substantial.

Acknowledgements.

We also thank Tomas Caithaml for implementing the first version of the TBP library, which is used to parse TBP specification.

8. REFERENCES

[1] L. de Alfaro and T. Henzinger. Interface automata. In Proceedings of the 8th European software engineering conference (ESEC/FSE-9). 2001. ACM, New York, NY, USA, 109-120.

[2] T. Bolognesi and E. Brinksma, Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems, vol. 14, no. 1, pp. 25-59, 1987.

[3] C. A. R. Hoare, Communicating sequential processes, Communications of the ACM, vol. 21, no. 8, pp. 666-677, 1978.

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

[5] BeJC, `http://d3s.mff.cuni.cz/projects/formal_methods/bejc`

[6] CoCoME : Common Component Modeling Example, `http://www.cocome.org`

[7] L. Bulej, T. Bures, T. Coupaye et al. CoCoME in Fractal, LNCS, vol. 5153, 2008.

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

[9] Component Reliability Extensions for Fractal Component Model, Project of Institute of Computer Science Academy of Sciences of the Czech Republic & France Telecom, 2004, `http://kraken.cs.cas.cz/`

[10] The Extended Static Checker for Java version 2, `http://kind.ucd.ie/products/opensource/ESCJava2/`

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

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

[13] P. Jancik. Checking Compliance of Java Implementation with TBP Specification, Master Thesis, Charles University, 2010. `http://d3s.mff.cuni.cz/publications/download/2010-Jancik-MasterThesis.pdf`

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

[15] J. Kofroň, P. Jančík, and P. Parízek. Refinement between TBP and Java Implementation of Components, Technical Report 2011/05, Charles University in Prague, 2011.

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

[17] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, In SEW-30, 2006.

[18] T. Poch. Towards Thread Aware Component Specifications, Ph.D. thesis, Charles University in Prague, Czech Republic, 2010.

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

[20] R. Allen and D. Garlan. A Formal Basis for Architectural Connection, ACM Trans. on Soft. Eng. and Meth., July 1997.

[21] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes, LNCS, vol. 6605, 2011.

[22] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP, IEEE International Conference on Software Engineering. and Formal Methods, 2006.

[23] N. E. Beckman, D. Kim, and J. Aldrich. An Empirical Study of Object Protocols in the Wild., European Conference on Object-Oriented Programming, 2011

[24] T. Bures et al.: Runtime support for advanced component concepts, in Proceedings of SERA 07, Busan, Korea, 2007

[25] E. Bruneton et al. An Open Component Model and Its Support in Java. Proceedings of Component-based Software Engineering, 2004