Checking Fractal Component Behavior Using Behavior Protocols*

J. Kofron[1,2], J. Adamek[1,2], T. Bures[1,2], P. Jezek[1], V. Mencl[3,2], P. Parizek[1] and F. Plasil[1,2]

1) Academy of Sciences of the Czech Republic, Czech Republic,
{kofron, adamek, bures, plasil}@cs.cas.cz
2) Charles University in Prague, Czech Republic,
{kofron, adamek, bures, jezek, mencl, parizek, plasil}@nenya.ms.mff.cuni.cz
3) United Nations University IIST, Macao, mencl@iist.unu.edu

1 Introduction

Building applications from of-the-shelf pieces, like libraries and components, is a promising approach to the future of software development. As such pieces are a subject to reuse, they have to have their interfaces and semantics clearly defined. To combine components from various vendors, one need a way to reason about component compatibility. Our experience with distributed component-based application has shown that reasoning about component compatibility based only on comparing component types (interface/method signatures) is not sufficient. Therefore, a kind of component semantic/behavioral specification is necessary.

Fractal [1] provides a hierarchical component model, where components may be nested (forming *composite* components) and only the lowest-level (primitive) components are implemented in a programming language. The architecture of a component application is described in the *Fractal Architecture Description Language* (ADL). For each component, its behavioral specification may be stated in the application ADL file.

In this paper, we describe results of the work on the "Component Reliability Extensions for Fractal Component Model" project [7] funded by France Telecom. The goal of the project is to extend the Fractal Component model and its Julia implementation with support for behavior protocols [8]. The paper is structured as follows: In Sect. 2, behavior protocols as a way of specifying component behavior are described, while Sect. 3 introduces various types of compatibility checks. Sect. 4 provides evaluation and concludes the paper.

2 Behavior Protocols

Behavior protocols are a platform for component behavior description. By the term *component behavior* we mean the traffic on its provided and required interfaces. The set of provided and required interfaces of a component forms the *frame* of the component and the behavior protocol belonging to this frame is called the *frame protocol*.

Behavior protocols are expressions generating a language of finite traces; each "word" describes a particular allowed trace of component behavior. Each trace is composed of events denoting one of the following actions:

- emitting a request of the method m on the interface i: !i.m↑
- accepting a request of the method m on the interface i: ?i.m↑
- emitting a response of the method m on the interface i: !i.m↓
- accepting a response of the method m on the interface i: ?i.m↓

These events are combined using both regular and special operators; special operators include, e.g., the *and-parallel* operator generating all interleavings of the operands' traces. As an example consider a simple component application at Fig.1. Behavior protocols describing allowed traces are:

* This work was partially supported by France Telecom under the external research contract number 46127110.

Component	Behavior protocol
LoggingDatabase	?db.start; (?db.get + ?db.put)*; ?db.stop
Database	?db.start{!lg.start}; (?db.get{!lg.log} + ?db.put{!lg.log})*; ?db.stop{!lg.stop}
Logger	?lg.start; ?lg.log*; ?lg.stop

We illustrate the meaning of behavior protocols on the *Database* component. Its protocol says that the *Database* component is able to accept only a `start` method call on its `db` interface at the beginning, which results in a `start` method call on the `lg` interface before the `start` call made on the *Database* component returns. After that, the component is able to receive any arbitrary number of `get` and `put` calls that results each time in a `log` method call on the interface `lg`. At the end, the *Database* and the *Logger* components are stopped via a `stop` method call in a similar way.

Having described all components with behavior protocols, we can reason about behavior compatibility.

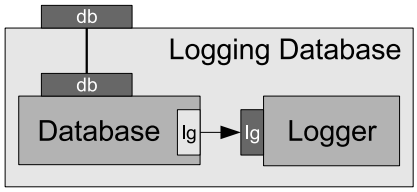

Fig. 1. A simple logging database component

3 Types of Checks

There are three types of compatibility and conformance checks performed before and after application is run — *static protocol check*, *JPF check*, and *runtime check*. Static and JPF checks are intended to be used before the application is run; these two checks are considered as verification (from a formal point of view), whereas runtime check is a kind of testing.

3.1 Static Protocol Check

This type of test provides complete information about behavior compatibility of application components on the level of abstraction behavior protocols provide. The static check employs two kinds of compatibility testing — compliance tests and composition tests[1]. Both tests are realized by applying a special composition operator called *consent* [5] to compose behavior protocols. The consent operator is able to capture three types of composition errors: *bad activity* denoting the situation when a component emits a method request and there is no component able to accept such a call at that moment, *no activity* denoting a deadlock, and *divergence* denoting a livelock, i.e. the situation when the communication trace forms a cycle without a chance to terminate (reach an accepting state).

The goal of the composition test is to verify absence of composition errors in communication of components on a particular level of nesting; the question expressing the meaning of this test would be "do all the siblings (components on a particular level of nesting) cooperate without errors?" In the example above, this test would employ compatibility checking of communication between the *Database* and *Logger* components.

The other kind of static check is the compliance test. It compares the specified behavior of a component with its first-level-of-nesting subcomponents; the question expressing the meaning of this test would be "do the children (subcomponents) do what their parent (the supercomponent) expects?" Here, this test would

[1] The definition of the compliance relation can be found in [8].

compare the behavior of the *LoggingDatabase* component with the behavior of the *Database* and *Logger* components[2].

Both the compliance and composition checks are done only at the level of behavior protocols, with no implementation code involved. Consequently, these tests can (and should) be done while the application is being designed, before implementation activities start. This allows to identify design errors at an early development stage, and to avoid the high costs of fixing the errors at a later stage.

As this type of check is based on exhaustive traversal of the state space of the behavior protocols, having successfully passed compliance and composition tests on all components and their subcomponents ensures the absence of communication errors within the entire component application. Obviously, this holds only if all the primitive components behave according to their frame behavior protocols. Verifying this relation is subject of the JPF check.

3.2 JPF Check

This test is used to ensure that the implementation of a primitive component is compliant to the behavior protocol. To perform it, we used Java PathFinder [2] in combination with the behavior protocol checker used in the previous type of check [9]. Java PathFinder (JPF) is a model checker for Java programs; it is based on a modified Java virtual machine that performs all possible program executions with respect to thread interleaving and non-deterministic choices. Nondeterminism is modeled based on assumptions about values returned by the random value methods provided by the JPF `Verify` class; this also allows to use non-deterministic (random) values to, e.g., simulate interaction with a user.

To check whether the implementation is compliant with the behavior protocol, JPF notifies the behavior protocol checker about instructions representing method invocations and returns from methods. Behavior protocol checker traverses the behavior protocol state space along the path corresponding to the JPF instruction notifications received. If an instruction cannot be executed in the protocol state space, behavior protocol checker reports a protocol violation. Similarly, if JPF reaches an end state (i.e. state with no further transitions), but the behavior protocol checker is not in an end state, a protocol violation is reported as well.

As JPF only accepts a complete program as its input, and the entire application is usually too large to be handled by JPF and behavior protocol checker, component environment has to be created to test each component separately. This issue is addressed in [3].

3.3 Runtime Check

When applied to all components of a Fractal application, combination of static and JPF checks provides formal verification of composition correctness and absence of composition errors. Sometimes, however, the complexity of a component hinders the use of these checks due to the enormous size of the component state space. In such situations, runtime checks may still provide useful piece of information about protocol violations.

Runtime check is similar to the JPF check, the key difference is that the implementation is executed by a standard virtual machine, and only a single trace corresponding to the actual run is checked in one execution (note that one can change some program input parameters to perform more thorough testing, but cannot assure complete code coverage). The advantage of this kind of check is that all the components can be tested simultaneously and there is no need to create component environments.

4 Evaluation and Conclusion

We have implemented a tool performing all three types of compatibility checks and tested it on a real-life component application. The application was a rather complex system allowing clients of various air-carriers

[2] Technically, the compliance test is performed by inverting the frame protocol of the containing component (swapping request and responses event modifiers), treating this component as another "sibling", and applying the composition test.

to access the Internet from airport lounges via local Wi-Fi networks. The whole Wireless Internet Access application was composed of about 20 Fractal component; more details can be found in [4] and [7]. The design of the application, i.e. creating the application architecture and writing and debugging the protocols, took a couple of months; but, it is worth to mention, as this was our first real-life Fractal application, the time usually spent on the specification of such an application is expected to be shorter. Having the bug-free specification in behavior protocols, the simplified implementation of the primitive components in Java was done in a straightforward way.

Even though we consider the aforementioned application quite complex, we successfully applied the static and JPF checks on all the application components and verified thus both the correctness of the communication and compliance of the implementation of the primitive components to their specification. Both types of check usually took several hours of machine time, which we still believe to be a reasonable time for these kinds of test. Note that when the specification contained an error, it was usually discovered very fast. Conversely, the runtime checks meant no significant time overhead in comparison with a normal application execution, as the protocol state space was generated on-the-fly quite efficiently. In addition, we implemented a method for determining what parts of a behavior protocol were actually used in a particular application run, which helped us to identify the typical program execution path and what parts of specification (and implementation) might be redundant.

References

1. E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J-B. Stefani. An Open Component Model and Its Support in Java. 7th SIGSOFT International Symposimum on Component-Based Software Engineering (CBSE7), LNCS 3054, Edinburgh, Scotland, May 2004.
2. Java PathFinder. http://javapathfinder.sourceforge.net
3. Parizek, P., Plasil, F.: Specification and Generation of Environment for Model Checking of Software Components. Tech. Report No. 2005/5, Dep. of SW Engineering, Charles University, Nov 2005
4. Jezek, P., Kofron, J., Plasil, F.: Model Checking of Component Behavior Specification: A Real Life Experience, Published in Preliminary Proceedings of International Workshop on Formal Aspects of Component Software (FACS'05), Macao, October 24-25, 2005, UNI-IIST Report No. 333, Oct 2005
5. Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis. Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363-377, DOI: 10.1002/smr.321, Online ISSN: 1532-0618, Print ISSN: 1532-060X, Sep 2005
6. JULIA framework (fractal implementation). http://fractal.objectweb.org
7. Component Reliability Extensions for Fractal Component Model. http://kraken.cs.cas.cz/ft/public/public_index.phtml
8. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
9. Parizek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Making Java PathFinder Cooperate with Behavior Protocol Checker, Tech. Report No. 2006/2, Dep. of SW Engineering, Charles University, Jan 2006