

Executable and Symbolic Conformance Tests for Implementation Models (Position Paper)

Thomas Baar

Universität Karlsruhe, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
Email: baar@ira.uka.de

Abstract. Following the Model-Driven Architecture (MDA), a system description consists of several models, i.e. views of the system. This paper is concerned with formal conformance tests between different models. It stresses the need for formal semantical foundations of all languages that are used to express models. In particular, we classify conformance tests for implementation models.

The Model-Driven Architecture (MDA) [14, 2] is the new strategy of the OMG to maintain the whole lifecycle of a system. A system is specified by a collection of several views, which concentrate on certain aspects and hides all non-relevant details. In the terminology of MDA a view is called *model*. Different kinds of models are usually expressed in different suitable languages. The implementation of a system is seen as just one model among others. MDA does not stipulate the usage of particular languages to express particular models. However, implementation models are usually expressed in terms of a programming language.

Especially in the development phase of a system, periodical checks on the consistency of all models are crucial. MDA calls this activity *conformance test* between two or more models. In this paper, we focus on conformance tests which can be performed by a machine. Obviously, such conformance tests presuppose a formal underpinning of all languages used to express the involved models.

We concentrate on the conformance test between the implementation model and a single additional model. For practical reasons, we assume Java to be the language for expressing the implementation model. The semantics of Java was originally defined in [3]. Due to its informal nature, [3] cannot be used directly for machine-based conformance tests. Instead, another representation of Java's semantics has to be chosen. We discuss two representations, and investigate the influence of that choice on the nature of the corresponding conformance tests.

Java semantics given by a compiler A pragmatic and popular approach defines the semantics of Java by the actual behaviour of programs on a machine.

Adopting this approach for the semantical foundation of implementation models means to define implementation models in terms of the behaviour of a *Black Box* that consists of the Java byte-code compiler and the Java Virtual Machine (JVM). This is fully acceptable but imposes a serious restriction for conformance tests. Obviously, conformance tests can be only performed against those models which rely semantically on the same *Black Box* and therefore are written in Java as well.

For large-scale projects, the development of unit tests [6] has proven to be a good practice. A suite of unit tests can be considered as a model which is different from the implementation model but written in the same implementation language. For instance, let us consider the following implementation:

```
public class Foo{  
    public int addTwo(int i){  
        return i+2;  
    }  
}
```

Assume, we want to test the conformance between the implementation model of class `Foo` and another model, that requires the result of operation `addTwo` to be always greater than the argument. The intended model can be (partially) expressed by some unit tests, e.g.

```
aFoo.addTwo(3) > 3  
aFoo.addTwo(5) > 5
```

The conformance test between both models is done simply by the automatic execution of the two test cases. Therefore, we name such conformance tests *executable conformance tests*.

The example reveals the big disadvantage of models given in a unit-test style. Here, the language Java is abused as a specification language with very limited expressive power, i.e. Java is unable to express abstract properties of a system. In the above example, the intended requirement – the return value of `addTwo` is always greater than the argument – can only be formulated approximately, by asserting it for a few arguments.

Abstract models, i.e. models which describe more abstract properties of the system, are therefore better formulated using a more suitable specification language than Java, e.g. OCL [13] or JML [8]. However, at a first glance, we have to sacrifice the conformance tests between abstract models and implementation models because the semantical descriptions of the specification languages OCL and JML do not rely on the *Black Box* used so far for the semantical foundation of Java.

We can get rid of that problem by changing the semantics of implementation models.

Java semantics in terms of logic There is quite a number of logical systems [11, 4, 12, 1] which (partially) capture the semantics of Java¹ formally. Furthermore, each of the logical systems is supported by the tools Isabelle [5], Loop-Tool [9], LOPEX [10], KeY-System [7], respectively. The support by sophisticated tools makes it feasible to check mechanically certain properties of Java programs.

Suppose, the implementation model is semantically based on the logical system described in [1] (Dynamic Logic). Then, conformance tests are enabled against all those models which are formulated within a language *similar* to the language of Dynamic Logic. Two languages are called *similar* iff they are defined on comparable semantics. For instance, OCL is similar to Dynamic Logic, and JML is similar to the logical system given in [4], since there are implemented translations integrated in the KeY-System and the Loop-Tool, respectively.

The languages OCL and JML are much more suitable than Java to describe abstract properties. For the `addTwo`-example, the following OCL-constraint formalises the intended requirement:

```
context Foo : addTwo(int i)
post:      result > i
```

A conformance test between this model and the implementation model can be processed by the KeY-System fully automatically. Internally, the KeY-System manipulates symbols such as `i`, `result`, etc. Therefore, we name such conformance tests *symbolic conformance tests*.

Conclusion

The MDA advocates the description of a system as a collection of several models. However, the languages to formulate the models are left open. This paper argues that machine-based conformance tests are only feasible between those models, which rely semantically on comparable definitions. We propose a classification of conformance tests for implementation models.

Conformance tests against the implementation model can be classified based on the style of semantical description for the programming language.

In a first case, we considered the programming language be semantically given by a reference to its compiler. Then, the other model used in the conformance test must semantically be given by a reference to the same compiler. Thus, both models and the conformance test are executable.

In the second case, the programming language is semantically given in terms of logical systems. This enables conformance test between the implementation model and a second model written in an expressive specification language, e.g.

¹ Unfortunately, some advanced concepts of Java, e.g. threads, reflection, are still excluded.

OCL, JML. The conformance test itself can be carried out using tools such as the Loop-Tool or the KeY-System. Since all such tools work at a symbolic level, user interactions are sometimes required. However, user interaction can be reduced to a minimum by the usage of a mature tool.

Acknowledgements

My thanks are due to Martin Giese, Reiner Hähnle, and Bernhard Beckert for their comments on earlier drafts of this paper.

References

1. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
2. D. D'Souza. Model-driven architecture and integration – opportunities and challenges, 2001. Available from www.kinetium.com.
3. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.
4. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Berlin, Germany*, LNCS 1783. Springer, 2000.
5. Isabelle. *Generic Theorem Proving Environment. Web pages*. Cambridge University and TU Munich. Available from www.cl.cam.ac.uk/Research/HVG/Isabelle.
6. JUnit. *Web pages*. Available from www.junit.org/index.htm.
7. KeY Project. *Integrated Deductive Software Design. Web pages*. University of Karlsruhe and Chalmers University Gothenburg. Available from i12www.ira.uka.de/~key.
8. G. T. Leavens, K. R. M. Leino, C. Ruby, and B. Jacobs. JML: Notations and tools supporting detailed design in Java. In *OOPSLA'2000 Companion*. ACM, 2000.
9. Loop Project. *Logic of Object-Oriented Programming. Web pages*. Katholieke Universiteit Nijmegen. Available from www.cs.kun.nl/~bart/LOOP.
10. LOPEX Project. *Logic-based Programming Environments. Web pages*. FernUniversität Hagen. Available from www.informatik.fernuni-hagen.de/import/pi5/forschung/lopex.html.
11. D. v. Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS 1523. Springer, 1999.
12. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
13. Rational Software Corp. et al. *Unified Modelling Language Semantics, version 1.3*, June 1999. Available from www.rational.com/uml/index.jttml.
14. R. Soley. Model driven architecture, 2000. White paper. Available from www.omg.org/mda.