Java Specification Extension for Automated Test Development

Igor B. Bourdonov, Alexey V. Demakov, Andrew A. Jarov,
Alexander S. Kossatchev, Victor V. Kuliamin, Alexander K. Petrenko, and
Sergey V. Zelenov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{igor,demakov,jandrew,kos,kuliamin,petrenko,zelenov}@ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract. The article presents the advantages of J@va, a specification extension of the Java language, intended for use in automated test development. The approach presented includes constraints specification, automatic oracle generation, usage of FSM (Finite State Machine) model and algebraic specifications for test sequence generation, and specification abstraction management. This work stems from the ISPRAS results of academic research and industrial application of formal techniques [1].

1 Introduction

The last decade has shown that the industrial use of formal methods became an important new trend in software development. Testing techniques based on formal specifications occupy a significant position among the most useful applications of formal methods. However, several projects carried out by the RedVerst group [3, 12] on the base of the RAISE Specification Language (RSL) [6] showed that the use of specification languages like RSL, VDM or Z, which are unusual for common software engineer, is a serious obstacle for wide application of such techniques in industrial software production. First, the specification language and the programming language of the target system often has different semantics and may even use different paradigms, e.g., one can be functional and the other can be object-oriented, so a special mapping technique must be used for each pair of specification and target language. Second, only developers having special skills and education can efficiently use a specification language. The possible solution of this problem is the use of specification extensions of widely used programming languages.

This article presents J@va – a new specification extension of Java language. Several specification extensions of programming languages and Java in particular already exist. ADL [5, 7] and iContract [8, 9] are the most known of them. A few extensions have been used in industrial projects. Why we invent a new one?

Our experience obtained in several telecommunication software verification projects shows that the formal testing method used in industry should not only allow automated test generation but also possess features such as clear modularization, suitable abstraction level management, separate specification and test design, and the support of test coverage estimation based on several criteria [14]. These subjects did not receive sufficient attention in ADL and iContract languages and test development technologies based on them. The absence of integrated solution explains the limited use of specification based methods in general and these languages and related tools particularly.

Every industrial test development technology should provide the answers on the following three questions.

- How to determine whether the component of the target system behaves correctly or not?
- How to determine the set of test cases, which makes the test complete in the sense that any additional test case can not add any important information? And how to estimate the results of the test, which does not contain all of these test cases for some reasons?
- How to organize the sequence of target operations calls during the test, *the test sequence*, to obtain the necessary test cases in the most effective way?

The first problem is usually solved with the help of the component's *oracle,* which can be generated from the specification of the component. Although oracle generation techniques are known (see, for example, [3–5]), they are not widely used in industrial projects. KVEST project [3] performed by our group is the largest example of such a project in the European formal methods database [2]. We do not consider the issues of automated oracle generation in this paper to comply with size restrictions. We refer the interested reader to the previously mentioned works [3–5].

The solution of the second problem is very important for the industry. Usually the solution is based on *test coverage criterion,* which gives the numerical measure of the test effectiveness. Coverage criteria based on the structure of the target code are widely known and used in the industry, but they are not sufficient. Such criteria show what part of target code is touched by the test. The important issue is also what part of functionality is touched by it. This problem can be solved with the help of *specification based coverage criteria.*

The third problem mentioned above is addressed by FSM based testing technique used in our approach called UniTesK technology (see [11] for details). The approach uses FSM model of the unit under test. Such a model is developed on the base of the coverage criterion chosen to obtain. Then, the FSM developed is traversed, and, so, the target coverage is achieved. We call the description of this model *the test scenario.* Test scenarios directly deal with test design while specifications describe the abstract functionality of target system.

2 Key Features of J@va Approach

In this section we present the goals of J@va design and J@va key features, explain their advantages, and compare them with ADL and iContract.

During the design of J@va language we tried to preserve the main features of our test development method, which is based on several previous successful projects (see [3]). These features are as follows.

- Automatic generation of test oracles on the base of specifications presented as pre- and postconditions of target operations.
- The possibility to define test coverage metrics and automatic tracking of coverage obtained during the test.
- Flexible FSM based testing mechanism.
- Dynamic test optimization for the target coverage criterion.

Along with that we must simplify and clarify the method to make it applicable in the software industry, not only in the research community.

Below we pay more attention to features not supported or supported insufficiently by J@va contenders. In particular, we do not consider the general software contract specification approach used in all mentioned languages [8, 13] and methods to specify exceptional behavior. Parallel processing specification and testing methods are also out of the scope of this article.

Specification of object state. J@va specifications are structured similar to Java code. The specification of the behavior of some component is presented as a specification class, which can have attributes that determine the model state, *invariants* representing the consistency constraints on the state of an object of this class, and specification methods representing the specifications of operations of the target component. Specification method can have pre- and postcondition. In ADL and iContract specifications are also presented as pre- and postconditions, but they have no special constructs for state consistency constraints. The same effect can be obtained only by including such constraints into pre- and postconditions of all class methods.

Axioms and algebraic specifications. J@va provides constructs to express arbitrary properties of the combined behavior of target methods in an algebraic specification fashion. The semantics of J@va axioms and algebraic specifications is an adaptation of the semantics of RSL ones [6]. Axioms and algebraic specifications serve as a source for test scenarios development – they are viewed as additional transitions in the FSM testing model. During testing we call the corresponding oracle for each method call in an axiom and then check the global axiom constraint. Similar constructs can be found only in specification languages and are absent in specification extensions as ADL and iContract.

Test coverage description. This is an essential feature for testing and software quality evaluation. Test coverage analysis also helps to optimize the test sequence dynamically by filtering the generated test cases, because usually there is no need to perform a test case that does not add anything to the coverage already obtained. Each coverage element can have only one corresponding test case. The coverage consisting of domains of different behavior, called the *specification branch coverage,* can be derived automatically from the J@va postcondition structure. J@va also has several special constructs for explicit test coverage description. The explicit coverage description and functionality coverage derivation allow providing fully automatic test coverage metrics construction and test coverage analysis. Neither ADL nor iContract has facilities for test coverage description and analysis.

Abstraction level management. The ability to describe system on different abstraction levels is very important both in forward and reverse engineering of complex systems. The support of abstraction level changing allows developing really implementation-independent specifications, whether we follow top-down design or bottom-up reverse engineering strategy. In J@va, specifications and source code are fully separated. Their interaction is provided by a special *binding code.* This code performs synchronization of the model object state with the implementation object state and translates a call of model method into a sequence of implementation methods invocations. It is necessary, because test sequence is defined on the model level in our method. This approach allows using one specification with several source code components and vice versa, it also ensures the modularity of specifications and makes possible their reuse. No other of known Java specification extensions provides such a feature. Larch [10] provides the infrastructure the most similar to the J@va one but supports only two-level hierarchy.

Test oracle generation. This is a standard feature of specification extensions intended to be used for test development. J@va, as ADL and iContract, supports automatic generation of test oracles from the specifications.

Test scenarios. Test scenarios provide the test designer with a powerful tool for test development. The scenarios can be either completely user-written or generated on the base of once written templates and some parameters specified by test designer. In general, a J@va scenario defines its own FSM model of the target system, called *the testing model.* A scenario defines the state class for this model and the transitions, which must be described in terms of sequences of target method calls. The testing model should represent a FSM, which can be obtained from the FSM representing the target system by removing some states and transitions, combining a sequence of transitions into one transition and subsequent factorization. One can find details of this approach, some methods and algorithms of testing model construction in [11], where they are formulated in terms of FSM state graph properties.

In a more simple case, test scenario represents the sequence of tested operation calls that can lead to some verdict on their combined work. The test constructed from such a scenario executes the stated sequence and assigns the verdict; it also checks the results of each operation with the help of the operation's oracle.

J@va allows the use in scenarios such constructions as iterations, nondeterministic choice and serialization. Iterations help to organize test case generation for one target operation. J@va test scenario is represented as a class with special methods – so-called scenario methods, which represent the transitions of the model.

Among existing Java extensions, only ADL provides some constructs for test case generation. However, complex tests, e.g. for a class as a whole, have to be written entirely in the target programming language. An essential shortcoming of this approach is the lack of state-oriented testing support that forces the test designer to spend considerable effort to ensure the necessary test coverage.

Open OO verification suite architecture. The verification suite consists of specifications, test scenarios, binding code, and Java classes generated from the specifications and the test scenarios. The set of classes and relations between these classes and between verification classes and target Java classes are well defined. The architecture is described in UML and is easy to understand by any software engineer having experience in design and development using Java language. The openness of the architecture does not mean necessity of the generated code customization for optimization or other purposes. There are other well-defined flexible facilities for fitting the verification suite. However the openness significantly facilitates the understanding and the use of the technology as a whole. ADL and iContract users could read (and reverse engineer) generated code, however the structure of generated test harness is considered a private issue of ADL/iContract translator and can be changed at any time.

Example of J@va specifications. Here we give an example of J@va specifications for bounded stack class with non-null elements. This example demonstrates some of the features itemized above.

```
specification package ru.ispras.redverst.se.java.examples.stack;
import java.util.Vector;

class StackSpecification {
  static public int MAX_SIZE = 2048;
  public Vector items = new Vector(MAX_SIZE); // model state

// object intergity constraint
  invariant I1()
  {
    return items.size() >= 0 && items.size() <= MAX_SIZE;
  }

// specification of pop() operation
  specification public synchronized Object pop()
    updates items.? // changes data available through items field
  {
    pre { return items.size() != 0; }
    post
    {
      branch "Single branch"; // defines single coverage element
      Vector old_items = items.clone();
    // method identifier refers to the result of operation
      old_items.addElement(pop);
    // @<expression> denotes the value of expression in pre-state
      return old_items.equals(@items.clone());
    }
  }

// specification of push() operation
  specification synchronized void push(Object obj)
    reads obj, updates items.?
  {
    pre { return obj != null && items.size() != MAX_SIZE; }
    post
    {
      branch "Sinlge branch";
      Vector new_items = @items.clone();
      new_items.addElement(obj);
      return items.equals(new_items);
    }
  }

// algebraic specifications
  equivalence synchronized Object push_pop(Object obj)
  {
    pre { return items.size() != MAX_SIZE; }
    alternative { push(obj); return pop(); }
    alternative { return obj; }
  }

  equivalence synchronized void pop_push() {
    pre { return items.size() != 0; }
    alternative { push(pop()); return; }
    alternative { return; }
  }
}
```

3 Conclusion

To become applicable in industrial software production, an automated test development technology must support a set of features that constitute something like a critical mass. The critical mass should be not too huge to be introduced in real-life software engineering and at the same time it should be sufficient for usual needs of software engineers. The J@va tries to achieve this goal. More detailed description of J@va and J@va based technology are presented on our web site [1].

References

1. http://www.ispras.ru/~RedVerst/
2. http://www.fmeurope.org/databases/fmadb088.html
3. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. *FM'99: Formal Methods. LNCS,* volume 1708, Springer-Verlag, 1999, pp. 608–621.
4. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering,* 24(3):161–173, 1998.
5. M. Obayashi, H. Kubota, S. P. McCarron, L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.xopen.org/exgr/icse/icse98.htm
6. The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
7. http://adl.xopen.org
8. R. Kramer. iContract – The Java Design by Contract Tool. // 4-th conference on OO technology and systems (COOTS), 1998.
9. http://www.reliable-systems.com/tools/iContract/iContract.htm
10. J. Guttag et al. The Larch Family of Specification Languages. // IEEE Software, Vol. 2, No. 5 (September 1985), pp. 24–36.
11. I. Bourdonov, A. Kossatchev, V. Kuliamin. Using FSM for Program Testing. *Programming and Computer Software,* Official English Translation of *Programmirovanie,* No. 2, 2000.
12. A. Petrenko, I. Bourdonov, A. Kossatchev, and V. Kuliamin. Experiences in using testing tools and technology in real-life applications. Proceedings of SETT'01, India, Pune, 2001.
13. B. Meyer. Object-Oriented Software Construction. Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1997.
14. A. K. Petrenko. Specification Based Testing: Towards Practice. In this transactions.