6-2007

Abstraction in Assertion-Based Test Oracles

Yoonsik Cheon
The University of Texas at El Paso, ycheon@utep.edu

Comments:

Technical Report: UTEP-CS-07-41

Abstraction in Assertion-Based Test Oracles

Yoonsik Cheon

TR #07-41
June 2007; revised August 2007

Keywords: abstraction, assertion, test oracle, runtime assertion checking, pre and postconditions, JML language.

1998 CR Categories: D.2.5 [*Software Engineering*] Testing and Debugging — Testing tools (e.g., data generators, coverage testing); F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

To appear in *First International Workshop on Software Test Evaluation (STEV 2007), Portland, Oregon, USA, October 11-12, 2007.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Abstraction in Assertion-Based Test Oracles

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968-0518
ycheon@utep.edu

Abstract

Assertions can be used as test oracles. However, writing effective assertions of right abstraction levels is difficult because on the one hand, detailed assertions are preferred for thorough testing (i.e., to detect as many errors as possible), but on the other hand abstract assertions are preferred for readability, maintainability, and reusability. As assertions become a practical tool for testing and debugging programs, this is an important and practical problem to solve for the effective use of assertions. We advocate the use of model variables—specification-only variables of which abstract values are given as mappings from concrete program states—to write abstract assertions for test oracles. We performed a mutation testing experiment to evaluate the effectiveness of the use of model variables in assertion-based test oracles. According to our experiment, assertions written in terms of model variables are as effective as assertions written without using model variables in detecting (injected) faults, and the execution time overhead of model variables are negligible. Our findings are applicable to other use of runtime checkable assertions.

1 Introduction

An assertion is a predicate or boolean expression, placed in a program, that should be always true at that place [12]. Assertions for runtime checking—assertions that are checked at runtime—become popular as a practical tool for testing and debugging programs [17]. If an assertion evaluates to false at runtime, it indicates that there is an error in the code for that particular execution, thus an assertion can be used as a test oracle and to narrow down a problematic part of the code [6]. C and C++ provide the `assert` macro, and Java 1.4 includes an `assert` statement. In design-by-contract languages such as Eiffel [15] and JML [13], assertions are used to codify the contracts (obligations and benefits) between the client and implementer of a program module, in the form of method pre and postconditions and class invariants.

When writing assertions, however, a programmer has a dilemma on the abstraction levels of the assertions. On the one hand, a detailed assertion is preferred to detect as many errors in code as possible. One the other hand, an abstract assertion is preferred for readability and maintainability; assertions too strongly tied to implementation details (e.g., data structures) might be hard to read and maintain, as even a small change in the implementation might require changes all over the assertions. It is not an easy task for the programmer to find the right abstraction level for the assertions.

We propose model variables as a solution to the programmer's dilemma on writing assertions. A *model variable* is a specification-only variable of which value is given in terms of program variables [7]. It allows a programmer to write assertions without directly referring to concrete program states. By using model variables, therefore, a programmer can write more easily assertions that are abstract, concise, and independent of representation details, and hence more readable and maintainable [7]. As the values of model variables are given by concrete program states, assertions written in terms of model variables can be evaluated and checked at run-time. In summary, model variables allow one to tune the level of abstraction in assertions.

The primary contribution of this paper is an evaluation of the effectiveness and efficiency of assertions written using model variables as test oracles. For this evaluation we performed a mutation testing experiment, and the results are very promising. The assertions written using model variables are as effective as the assertions containing no model variables in revealing injected faults without noticeable runtime overheads (see Section 4).

2 The Problem

In this section we explain the abstraction problem in assertions with an example written in JML [13], a formal behavioral interface specification language for Java. Figure 1

```
1  public class SimpleList {
2   /*@ spec_public @*/ private Object[] elems;
3   /*@ spec_public @*/ private int last;
4   //@ public invariant last >= -1 && last < elems.length;
5
6   /*@ requires obj != null;
7     @ assignable last, elems;
8     @ ensures last == \old(last + 1) &&
9     @   elems[last] == obj &&
10    @   (\forall int i; i >= 0 && i <= \old(last);
11    @     elems[i] == \old(elems[i]));
12    @*/
13  public void append(Object obj) { /* ... */ }
14
15  // the rest of definition
16 }
```

Figure 1. Class `SimpleList` annotated with assertions written in JML.

shows a partial definition of class `SimpleList` annotated with JML assertions. In JML, the behavior of a Java class is specified by writing class invariants (**`invariant`** clauses) and pre and postconditions (**`requires`** and **`ensures`** clauses) for the methods exported by the class. The pre and postconditions are viewed as a contract between the client and the implementer of the class. The client must call a method in a state where the method's precondition holds, and the implementer must guarantee that the method's postcondition holds after such a call. The postconditions, therefore, can be used as test oracles when testing the methods, and there are several tools that turn JML assertions into test oracles [6, 4, 5].

The `SimpleList` class implements an unbounded list by representing it as an array of objects (`elems`) and an index (`last`) that denotes the last element. The JML assertions are written in terms of private program variables such as `elems` and `last`, which are declared to be spec_public because they are used in the specification of public methods such as `append`. The **`assignable`** clause in line 7 states the frame condition allowing the `append` method to change the values of `last` and `elems`. The postcondition in lines 8–11 states the argument (`obj`) to be appended to the list, by constraining the new values of `elems` and `last` in terms of their old values; the built-in JML expression **`old`**(e) denotes the value of expression e in the pre-state, i.e., just before the method invocation.

What is wrong with the above assertions? There are several problems with directly referring to program variables in assertions. First, it exposes to clients the implementation details such as data structures that are irrelevant to the clients. In JML, this is often indicated by the use of spec_public fields, as in lines 2–3 of the example. It opens up the door for the client code to be tied to a particular implementation choice or decision, e.g., the use of an array to store the elements. Second, such assertions tend to be long and complicated, thus hard to read and understand, as in general changes to each program variable have to be documented as done in the postcondition of the `append` method. Third, such assertions are not reusable and hard to maintain, as they are tied to particular implementation choices and decisions and a small change in the implementation may necessitate changes throughout the assertions. This is partly because program variables now perform dual roles of providing a representation for the implementation and a vocabulary for writing the assertions. For example, if one decides to keep track of the number of elements in the list, instead of the index of the last element, the assertions in lines 4 and 7–11 have to be rewritten. Finally, the approach doesn't work for interfaces, as Java interfaces cannot contain program variables; however, interfaces are perfect places to add contracts between clients and implementers (see Section 3).

3 Our Approach

A *model variable* is a specification-only variable providing an abstraction mechanism for writing assertions [7]. The most distinctive feature of a model variable is that its value is given in terms of program variables by defining a mapping from concrete program states to abstract specification states. Thus, model variables allow one to specify program properties in a way that is not only abstract, concise, and independent of representation details but also can be checked at run-time.

We propose to use model variables to write abstract assertions for test oracles. For example, Figure 2 shows the specification of the `append` method rewritten using a model variable. The annotation in line 3 defines a public model field `seq`, that is used in the postcondition (line 7) of the `append` method. A list is now viewed abstractly and manipulated in assertions as a sequence of objects; A JML library class `JMLObjectSequence` defines an immutable sequence of objects for use in assertions. As before, a list is implemented as an array of objects (lines 11–12); the **`in`** annotations allow the methods that may change the model variable `seq` to also change the program variables `elems` and `last`. The **`represents`** clause in lines 14–15 defines the abstraction function for the model variable `seq` by mapping an array of objects to a sequence.

How does the use of a model variable solve the problems associated with directly referring to program variables in assertions? First, irrelevant implementation details and decisions such as data structures are not exposed to the clients. In the revised specification in Figure 2, there is no spec_public field, and both program fields and the **`represents`** clause are private. Only the model field is public, and it presents to the clients a list as a sequence of objects, thus hiding implementation details. Second, assertions are now succinct, concise, and abstract, thus increas-

```
1  //@ model import org.jmlspecs.models.JMLObjectSequence;
2  public class SimpleList2 {
3   //@ public model JMLObjectSequence seq;
4
5   /*@ requires obj != null;
6     @ assignable seq;
7     @ ensures seq.equals(\old(seq.insertBack(obj)));
8     @*/
9   public void append(Object obj) { /* ... */ }
10
11  private Object elems[]; //@ in seq;
12  private int last = -1; //@ in seq;
13
14  /*@ private represents seq <-
15    @  JMLObjectSequence.convertFrom(elems, last+1); @*/
16
17  // the rest of definition
18 }
```

Figure 2. Class `SimpleList2` with JML assertions written using a model variable.

ing clarity and improving understanding. For example, the `SimpleList2` class has about 20% less lines of assertions than the `SimpleList` class (see Section 4). In addition, the assertions are written in the vocabulary of clients (i.e., sequences) not in terms of a particular representation or implementation data structure. Third, assertions are now more reusable and easier to maintain, as they are not tied to particular implementation choices and decisions, and a change in the implementation is localized and does not have a ripple effect on the assertions. For example, if one wants to use a different data structure (e.g., a linked list instead of an array), then the only part of the assertions that needs to be changed is the definition of the abstraction function; the rest remains the same. This is partly due to a separation of the roles that program variables and model variables play. Finally, the approach does work very well for interfaces, as model variables can be defined in interfaces; model variables and other assertions such as invariants and method pre and postconditions of interfaces are inherited by implementing classes, and implementing classes only need to supply abstraction functions for the inherited model variables to make assertions written in terms of the model variables runtime checkable.

4 Evaluation

The use of model variables improves, among others, the readability, reusability, and maintainability of assertions, and thus assertion-based test oracles. Is it also effective and efficient? To answer this question, we performed a small experiment to measure quantitatively the effectiveness and efficiency of the use of model variables in assertions. We implemented an unbounded, array-based list with 8 methods including a constructor and a private helper method `doubleTheSize()`, such as `insert(int, Object)`, `append(Object)`, `remove(int)`, `get(int)`, `size()`, and `isEmpty()`, and documented its behavior with and without using a model variable, respectively. The one without a model variable is the class `SimpleList` and the one with a model variable is the class `SimpleList2`, both of which partial definitions are shown in the previous sections. The `SimpleList2` class has 20% less assertions than the `SimpleList` class in terms of source code lines, 38 vs. 48 lines of assertions for all 8 methods.

We first performed a mutation testing experiment to evaluate the effectiveness of the use of model variables in assertion-based test oracles. Mutation testing is based upon seeding a fault to a program and determining whether testing identifies the seeded fault. If a test case distinguishes between the mutated program (referred to as *mutant*) and the original program, it is said to *kill* the mutant. The objective of our mutation testing is to compare the effectiveness, as test oracles, of assertions written in terms of model variables against those written without model variables. By the effectiveness we mean the ability of an assertion in detecting faults—i.e., inconsistencies between the assertion and code. In our experiment, we first mutated the code, by introducing three mutation operations: value replacement that replaces a value with another value of the same type (e.g., true for false), operator replacement that replaces an operator with another, variable replacement that replaces a variable with another variable or value of the same type, and statement replacement that replaces a statement with another statement (e.g., empty statement). We seeded total 10 faults manually and performed random testing using JET. JET [5] is a fully automated unit testing tool for Java classes, automating each step of testing from test data generation to test execution and test result determination; e.g., for each method under test it generates a set of test cases randomly, each consisting of a receiver and arguments [8]. We used JET to generate 5 suites of random test data and export them as JUnit test classes so that we can run the same test suites for both (mutated) `SimpleList` and `SimpleList2` classes. To prevent interferences among seeded faults, we tested one mutant at a time.

Table 1 shows the result of our mutation testing. The average kill rates of `SimpleList` and `SimpleList2` are the same at 0.78, and every mutant killed by the first class was also killed by the second class, and vice versa.[1] Thus,

[1] In our initial experiment we found a mutant that survived an assertion of `SimpleList` but was killed by an assertion of `SimpleList2` written using a model variable. The first assertion has a quantifier while the second does not. It was soon revealed, however, that this was due to a deficiency in JML's instrumentation of quantifiers for runtime assertion checking. Specifically, runtime checking might be incomplete for a quantifier if its range is defined in terms of a modified post-state variable and its predicate contains an old expression. We rewrote all such quantifiers to eliminate post-state variables in the ranges of the quantifiers.

Table 1. Result of a mutation testing, where f is the number of injected faults (or mutants), and k_1, and k_2 are the numbers of mutants killed by `SimpleList` and `SimpleList2`, respectively.

test suite	no. of tests	no. of faults			det. ratio		
		inj. f	det. k_1	det. k_2	k_1/f	k_2/f	k_2/k_1
1	32	10	8	8	0.8	0.8	1.00
2	36	10	8	8	0.8	0.8	1.00
3	38	10	6	6	0.6	0.6	1.00
4	53	10	9	9	0.9	0.9	1.00
5	65	10	8	8	0.8	0.8	1.00
avg.					0.78	0.78	1.00

Table 2. Time efficiency of model variables. The elapsed times t_1, t_2, and t_3, respectively, are the total time required to execute the tests by `SimpleList`, `SimpleList2`, and `SimpleList` without runtime assertion checks.

test suite	no. of tests	elapsed time (sec)			ratio	
		t_1	t_2	t_3	t_2/t_1	t_1/t_3
1	68	0.21	0.21	0.02	1.00	10.05
2	86	0.25	0.26	0.02	1.04	12.05
3	92	0.22	0.23	0.03	1.06	7.33
4	113	0.32	0.32	0.05	1.00	6.40
5	127	0.34	0.33	0.04	0.97	8.50
avg.					1.01	8.87

the assertions written using a model variable are as effective as those written without using a model variable in killing mutants. We examined each of the surviving mutants and found that either the (randomly generated) test cases were inadequate to kill the mutant or the mutant was what is referred to as an equivalent mutant, a mutant that has the same observable behavior as the original program. For example, the private helper method `doubleTheSize` that increases the size of the array was mutated so as to increase the size even if the array is not full, and this mutant was never killed. We could have killed this kind of mutants by having multiple specifications of different visibility, as JML supports specification visibility; i.e., we could have added a private assertion that states that the size of the array is increased only if the array is full. However, this is a fundamental problem of an assertion-based test oracle in that a missing assertion cannot be tested for.

Is the use of model variables efficient in terms of runtime speed? To answer this question, we performed another random testing using JET. As before we exported randomly generated test data as JUnit test classes, but this time we measured the execution time of the test classes. Our experimental results are summarized in Table 2. As shown in the second to the last column (t_2/t_1), the execution time overhead of model variables are negligible; it is almost the same as that of assertions written without using model variables. However, the cost of JML's runtime assertion checking is very high, on average, at about 9 times slowdown in terms of the runtime speed.

5 Related Work

The basis of our work is the use of assertions as test oracles. The origin of this idea can be traced back to the use of formal specification as test oracles [11]. Antoy and Hamlet describe an approach to checking the execution of an abstract data type's implementation against its specification [1]. Their approach is similar to the technique of multi-version programming except that one version is an algebraic specification, serving as a test oracle. The algebraic specification is executed by term rewriting and is compared with the execution of the implementation. For the comparison the user has to provide an abstraction function that maps implementation states to abstract values. In assertion-based approaches such as JML, no separate specification program needs to run in parallel with the implementation.[2] Peters and Parnas proposed a tool that generates a test oracle from formal program documentation written in tabular expressions [16]. The test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. We believe that, with a suitable tool support, many assertion or design-by-contract languages such as Anna, APP, Eiffel, JML, and ADL/Java can be turned into assertion-based test oracle specification languages. For example, Cheon and Leavens employed the runtime assertion checker of JML as a test oracle engine [6], thus turning JML into an assertion-based test oracle language for Java. Our work enriches this idea further by promoting the use of model variables to write test oracle assertions that are abstract as well as detailed.

JML is unique in that there are no other assertion or design-by-contract languages that support model variables. The model variables of JML build on the work of Leino and Nelson [14] that clarified the semantics of specification-only variables, particularly with respect to frame conditions. They introduced specification-only variables to solve the problem of information hiding while still being able to specify and verify programs in a model-oriented style. Several assertion or interface specification languages such as J@va

[2] The multi-version approach can be simulated to some extent in JML by using ghost variables. A *ghost variable* is a specification-only variable that is manipulated in specifications by using specification statements such as the `set` statement [13].

[3], Abstract State Machine Language (AsmL) [2], and RESOLVE [10] support executable specifications written using abstract variables. However, the abstract variables are not associated with program variables through abstraction functions directly; instead, the mappings are typically given by (user-provided) binding code.

Coppit and Haddox-Schatz evaluated the effectiveness of specification-based assertions as test oracles [9]. The assertions were manually translated from Object-Z specifications, and their two case studies indicate that specification-based assertions can effectively reveal faults, as long as they adversely affect the program state. A similar conclusion can be made on assertions containing model variables because they are as effective as assertions containing no model variables.

6 Conclusion

A model variable allows one to tune the levels of abstraction in assertion-based test oracles with no or minimal additional cost associated with its use. A model variable is a specification-only variable of which abstract values are given by an explicitly-stated mapping (or function) from program state or variables. As shown by a mutation testing experiment, assertions written in terms of model variables can be as abstract as possible to hide implementation decisions and details from the client or tester, without losing the fault-detection capability. The execution time overhead of such assertions is also negligible.

We hope that model variables will facilitate the use of assertion-based test oracles, as they provide an important weapon to cope with two conflicting but key requirements of assertion-based test oracles, abstraction in oracle assertions and thoroughness of testing.

Acknowledgment

This work was supported in part by the National Science Foundation under Grant No. CNS-0509299 and CNS-0707874. Thanks to the STEV 2007 program committee for comments on an earlier draft of this paper.

References

[1] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, Jan. 2000.

[2] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.

[3] I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, and S. V. Zelenov. Java specification extension for automated test development. In *PSI '02: 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Akademgorodok, Novosibirsk, Russia, July 2–6*, pages 301–307, 2001.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[5] Y. Cheon. Automated random testing to detect specification-code inconsistencies. In *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice, July 9-12, 2007, Orlando, Florida, U.S.A*, pages 112–119, July 2007.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[7] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005.

[8] Y. Cheon and C. E. Rubio-Medrano. Random test data generation for java classes annotated with JML specifications. In *Proceedings of the 2007 International Conference on Software Engineering Research and Practice, Volume II, June 25–28, 2007, Las Vegas, Nevada*, pages 385–392.

[9] D. Coppit and J. M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, pages 305–314, Apr. 2005.

[10] S. H. Edwards, M. Sitaraman, B. W. Weide, and J. Hollingsworth. Contract-checking wrappers for C++ components. *IEEE Transactions on Software Engineering*, 30(11):794–810, Nov. 2004.

[11] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, Oct. 1969.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[14] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, Sept. 2002.

[15] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[16] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, Mar. 1998.

[17] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.