

# Automating Java Program Testing Using OCL and AspectJ

Yoonsik Cheon and Carmen Avila

TR #09-32  
October 2009

**Keywords:** pre and postconditions, random testing, runtime assertion checking, AspectJ, Object Constraint Language.

**1998 CR Categories:** D.2.2 [*Software Engineering*] Design Tools and Techniques — Modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract; D.2.5 [*Software Engineering*] Testing and Debugging — Monitors, testing tools; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

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

# Automating Java Program Testing Using OCL and AspectJ

Yoonsik Cheon and Carmen Avila  
Department of Computer Science  
University of Texas at El Paso  
El Paso, TX 79968

## Abstract

*Random testing can eliminate subjectiveness in constructing test data and increase the diversity of test data. However, one difficult problem is to construct test oracles that decide test results—test failures or successes. Assertions can be used as test oracles and are most effective when derived from formal specifications such as OCL constraints. If fully automated, random testing can reduce the cost of testing dramatically. In this paper we propose an approach for automating Java program testing by combining random testing and OCL. The key idea of our approach is to use OCL constraints as test oracles by translating them to runtime checks written in AspectJ. We realize our approach by adapting existing frameworks for translating OCL to AspectJ and assertion-based random testing. We evaluate the effectiveness of our approach through case studies and experiments. Our approach can detect errors in implementations and OCL constraints, as well, and provide a practical means for using OCL in design and programming.*

**Keywords:** pre and postconditions, random testing, runtime assertion checking, AspectJ, Object Constraint Language.

## 1 Introduction

Testing is laborious, time consuming, error-prone, and costly. Automated random testing can reduce the cost of testing dramatically by picking up an arbitrary input value from the set of all possible input values of the program under test. Random testing also has potential for finding faults that are difficult to find in other ways because it eliminates the subjectiveness in constructing test cases and increases the variety of them. There are random testing tools for Java [4, 6]. However, one difficult problem for a general use of random testing is to automate test oracles that determine test outcomes—test successes or failures.

Assertions can be used as test oracles [3, 5]; 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. It is said that assertions are most effective when they are derived or generated from the formal specification of a program.

The Object Constraint Language (OCL) [14] can be used to write such a formal specification. OCL is a textual notation to specify constraints or rules that apply to UML models such as class diagrams. It is based on mathematical set theory and predicate logic and can express relevant information about the systems being modeled that cannot otherwise be expressed by diagrammatic notations such as class diagrams. Using a combination of UML and OCL, one can build a precise design model that includes detailed design decisions and choices along with the semantics such as class invariants and operation pre and postconditions. There are approaches, frameworks, and support tools for Java to turn OCL constraints into runtime checks [2, 7].

In this paper we combine random testing and OCL to automate Java program testing. We use OCL constraints as decision procedures for filtering randomly selected test data and determining test results as well. To realize this idea, we adapt our earlier approach for translating OCL constraints to AspectJ runtime checking code [2]. In particular, we refine different types of OCL constraint violations so that they can be used as testing decision procedures. We also extend our automated, assertion-based testing tool for Java [4] to recognize OCL constraint violations. Our case studies and mutation testing experiments show that our approach is effective in detecting errors in implementations and in OCL constraints, as well. It also provide a practical means of using OCL in programming and testing.

The remainder of this paper is structured as follows. In Section 2 we give background information on OCL and AspectJ. In Section 3 we describe our approach by first illustrating it through an example and then explaining the details. In Section 4, we evaluate our approach through case studies and experiments. We conclude our paper with related work in Section 5 and concluding remarks in Section 6.

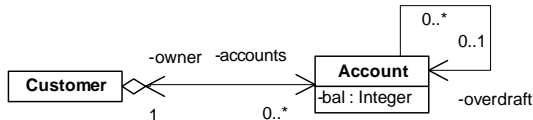


Figure 1. UML class diagram

## 2 Background

### 2.1 OCL

The Object Constraint Language (OCL) [14] is a textual, declarative notation to specify constraints or rules that apply to UML models. A UML diagram alone cannot express a rich semantics of and all relevant information about an application. The diagram shown in Figure 1, for example, is a UML class diagram for bank accounts. A customer can own several bank accounts, and an account can be linked to another account for overdraft protection. However, the class diagram doesn't express the fact that an account cannot be linked to itself for overdraft protection. It is very likely that a system built based only on the diagrams will be incorrect. OCL allows to precisely describe this kind of additional constraints on the objects and entities present in a UML model. It is based on mathematical set theory and predicate logic. The above-mentioned fact can be expressed in OCL as follows.

```
context Account
  inv: self <> overdraft
```

This constraint, called an *invariant*, states a fact that should be always true in the model. The keyword **self** denotes the object being constrained by an OCL expression, called a *contextual instance*; in this case it is an instance of the `Account` class. The invariant says that an account cannot be equal to its overdraft protection account. It is also possible to specify the behavior of an operation in OCL. For example, the following OCL constraints specifies the behavior of an operation `Customer::addAccount` by writing a pair of predicates called *pre* and *postconditions*.

```
context Customer::addAccount(acc: Account): void
  pre: not accounts->includes(acc)
  post: accounts = accounts@pre->including(acc)
  post: account.owner = self
```

The pre and postconditions states that, given an account not already owned by a customer, the operation should insert the account to the set of accounts owned by the customer. The postfix operator `@pre` denotes the value of a property (`accounts`) in the pre-state, i.e., just before an operation invocation. The constraints are written using OCL collection operations such as `includes` and `including`.

### 2.2 AspectJ

AspectJ [10] is an aspect-oriented extension for the Java programming language to address crosscutting concerns. A

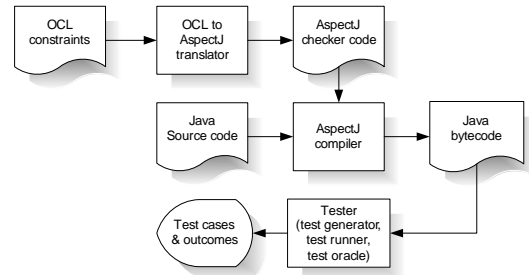


Figure 2. Automated testing using OCL

*crosscutting concern* is a system-level, peripheral requirement that must be implemented by multiple program modules, thereby leading to tangled and scattered code. Examples of cross-cutting concerns include logging, security, persistence, and concurrency. AspectJ provides built-in language constructs for implementing crosscutting concerns in a modular way. The key idea is to denote a set of execution points, called *join points*, and introduce an additional behavior, called an *advice*, at the join points. The following code shows an AspectJ aspect that checks the precondition of the `addAccount` method described earlier.

```
public privileged aspect PreconditionChecker {
  pointcut addAccountExe(Customer c, Account a):
    execution(void Customer.addAccount(Account))
    && this(c) && args(a);

  before(Customer c, Account a): addAccountExe(c, a) {
    if (c.accounts.contains(a))
      throw new RuntimeException("precondition_violation");
  }
}
```

The pointcut declaration designates a set of execution points and optionally exposes certain values at those execution points. The pointcut `addAccountExe` denotes executions of the `addAccount` method and exposes the receiver (`c`) and the argument (`a`). The **before** keyword introduces an advice that is to be executed before the execution of a join point; there are also *after* and *around* advices [10]. In the example, the advice is executed right before the execution of the `addAccount` method and checks its precondition by referring to the values exposed at that join point. Note that the aspect is declared to be *privileged*, which means that it bypasses Java language access checking and thus can access private members. This allows the aspect, for example, to access private fields such as `accounts` used in the precondition. If the `Customer` class is compiled with the above aspect, all invocations of the `addAccount` method that violate the precondition will be detected and result in runtime exceptions.

## 3 Approach

Figure 2 depicts our approach for automatically testing Java programs using OCL. The key idea of our approach is

to use OCL constraints for both filtering test data and determining test results. We accomplish this by turning OCL constraints into runtime checks. For this we translate OCL constraints to AspectJ code which is to be compiled with Java code under test to detect constraint violations at runtime. Our testing approach is dynamic in that we run the code under test to generate test data. We first generate test data randomly and then perform a test execution by invoking the method under test with the generated test data; we assume that each method of a class be tested separately. If the method invocation results in a pre-state constraint violation such as a precondition violation, we reject the test data as inappropriate for testing the method because such a constraint is the client’s obligation. If the invocation results in a post-state constraint violation such as a postcondition violation, we treat it as a test failure because such a constraint is the implementer’s obligation; otherwise, it is a test success. In summary, we do dynamic testing by generating test data randomly and using OCL constraints as test oracles.

In the following subsections we first illustrate our approach using a small example and then explain the details of translating OCL constraints to AspectJ runtime checks and performing dynamic testing based on runtime checks.

### 3.1 Illustration

Here we illustrate our approach by using the bank account example from the previous section. Our approach consists of two steps: test preparation and dynamic testing. The test preparation step prepares a Java program for testing by translating its OCL constraints to runtime check code written in AspectJ and compiling the program with the resulting AspectJ code. This enables us to detect OCL constraint violations at runtime. The dynamic testing step utilizes OCL constraint violations to filter test data and to determine test results. As an example, consider the following OCL constraint for the `withdraw` operation of the `Account` class.

```
context Account::withdraw(amt: Integer): void
pre: amt > 0 and amt <= bal
post: bal = bal@pre - amt
```

The test preparation step translates the above constraint to an AspectJ aspect, something like the following.

```
public privileged aspect AccountChecker extends OclChecker {
    pointcut withdrawExe(Account acc, int amt):
        execution(void Account.withdraw(int))
        && this(acc) && args(amt);

    void around(Account acc, int amt): withdrawExe(acc, amt) {
        checkPre(amt > 0 && amt <= acc.bal);
        int preBal = acc.bal;
        proceed(acc, amt);
        checkPost(acc.bal == preBal - amt);
    }
}
```

The aspect intercepts every execution of the `withdraw` method and checks its pre and postconditions specified as

OCL constraints. For this, it defines an *around* advice that surrounds a join point and thus can perform custom behavior before and after the *proceeding* to the join point. The framework class `OclChecker` defines several utility methods such as `checkPre` and `checkPost` for checking constraints and reporting constraint violations, e.g., by throwing an appropriate exception (see Section 3.2). The aspect is compiled along with the `Account` class to enable runtime constraint checking.

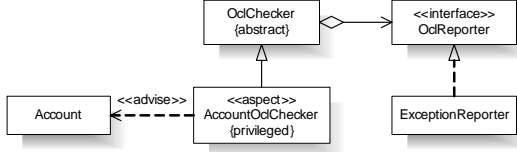
The next step is to perform dynamic testing. We first generate test data randomly, where each test case consists of a receiver and arguments. To test the `withdraw` method, for example, we need an `Account` object (receiver) and an `int` value (argument). Possible test cases include: `(new Account(100), -10)`, and `(new Account(100), 50)`, assuming that the `Account` class has a constructor that takes the initial balance as an argument. We then execute the tests by invoking the method under test with the generated test data, e.g., `new Account(100).withdraw(50)`. During the test execution we observe an occurrence of an OCL constraint violation. If we encounter a precondition violation, we reject the test data as inappropriate because the precondition is the client’s obligation. For example, the test case `(new Account(100), -10)` is rejected because it violates the precondition stating the positiveness of the withdrawal amount. If we encounter a postcondition violation, we interpret it as a test failure because the postcondition is the implementer’s obligation; otherwise, it is a test success. For example, the test case `(new Account(100), 50)` results in either a test success or failure.

### 3.2 Translating OCL to AspectJ

We adapt the approach proposed in [2] to translate OCL constraints to AspectJ aspects. The resulting aspect is called a *constraint checking aspect*. The approach is modular in that constraint checking aspects exist separate from the implementation code, which is oblivious of the aspects. When compiled with the aspects, however, the implementation is checked for OCL constraint violations at runtime.

Figure 3 shows the framework for checking OCL constraints at runtime. For each class we have a separate aspect that advises the class. This constraint checking aspect is responsible for checking all OCL constraints specified for the class. Each constraint checking aspect is defined to be a subclass of an abstract class, `OclChecker`. This class provides a set of utility methods, such as a mechanism for reporting constraint violations, to constraint checking aspects. It uses the strategy design pattern to separate constraint checking from violation reporting and to select a reporting mechanism appropriate for a particular application. In our case, a constraint violation is reported by throwing an exception, and this facilitates the testing framework to use OCL constraints as test oracles during dynamic testing (see Section 3.4).

As shown earlier, an OCL constraint is translated to point-



**Figure 3. OCL constraints checking framework**

cuts and advices. The pointcuts define execution points at which constraint checks are to be performed, and the advices perform actual constraint checks. It is also common that pointcuts expose values such as the receiver and arguments at the execution points so that they can be referred to by the constraint checking advices. The advices check OCL constraints and thus are often direct translations of the OCL constraints.

One adaptation in translating OCL to AspectJ is that we need to distinguish two different kinds of precondition violations. Let’s say that a client invokes a method, say  $m$ . If  $m$ ’s precondition is violated, this violation is called an *entry precondition violation*. On the other hand, if  $m$ ’s method body invokes another method, say  $n$ , of which precondition is violated, this violation is termed an *internal precondition violation*; it is internal from the  $m$ ’s client point of view. The reason for this distinction is that constraint violations are used as test oracles, and the later is a test failure while the former is not [3] (see Section 3.4). To support this distinction, we translate OCL constraints to *around* advices. For example, the pre and postconditions of the withdraw method are translated to the following advice.

```

/** Checks pre and postconditions of the withdraw method.
 * pre: amt > 0 and amt <= bal
 * post: bal = bal@pre + amt */
void around(Account acc, int amt): withdrawExe(acc, amt) {
    checkPre(amt > 0 && amt <= acc.bal);
    int preBal = acc.bal;
    try {
        proceed(acc, amt);
    } catch (OclEntryPreconditionError e) {
        throw new OclInternalPreconditionError(e);
    }
    checkPost(acc.bal == preBal - amt);
}

```

Note that the proceed call is enclosed inside a try-catch statement to catch and turn an entry precondition violation into an internal precondition violation.

### 3.3 Generating Test Data

We use the approach presented in [4] to generate test data automatically. That is, we generate test data randomly for each method of the class under test, say  $C$ . A test case thus is a tuple of objects and values,  $\langle r, a_1, \dots, a_n \rangle$ , where  $r$  is a receiver object of class  $C$  and  $a_i$ ’s are arguments. We use a different strategy to generate the element of a test case based on its type. For a primitive type, an arbitrary value of that

type is selected randomly. For an array type, the dimension is chosen randomly and then the elements are generated by applying the strategy of the element type.

For a class type, we cannot create an object of an arbitrary state directly because the object’s state is hidden. We do this indirectly by first instantiating a class and changing its state through a series of method calls. Let  $C$  be a class with a set of constructors,  $c_i$ ’s, and methods,  $m_i$ ’s. We classify them into four categories based on their signatures or suggestions from the user: basic constructor, extended constructor, mutator, and observer. A basic constructor creates a new instance of a class without needing another instance of the same class, whereas an extended constructor can create a new instance, given other instances of the class. A mutator changes the state of an object.

We represent an instance of  $C$  as a sequence of constructor and method calls,  $\langle s_0, s_1, \dots, s_n \rangle$ , where  $s_0$  is a basic constructor call and the rest are extended constructor or mutator calls. That is, to generate an instance of a class we first invoke one of its basic constructors and then, to change the instance’s state, invoke several extended constructors or mutators. The constructors and mutators are selected randomly. For example, below are three example call sequences for the `Account` class.

- 1:  $\langle \text{new Account}(10) \rangle$
- 2:  $\langle \text{new Account}(10), \text{deposit}(10) \rangle$
- 3:  $\langle \text{new Account}(10), \text{deposit}(20), \text{withdraw}(50) \rangle$

Note that not all instances—represented as call sequences—are *feasible*. Each call in the sequence has to satisfy the class invariant and method pre and postconditions. The last sequence, for example, fails to create an instance because the `withdraw` call fails to meet its precondition; the account has an insufficient balance (30) for the withdrawal request (50).

### 3.4 Executing Tests

Our approach is dynamic in that we run the method under test to generate its test cases. We run the method for two purposes: to filter out generated test cases and to determine test results. For both, we use OCL constraints as a decision procedure (see Figure 4). A generated test case is inappropriate for testing a method if it doesn’t satisfy the precondition of the method. Because the precondition is a client’s obligation, such a test case is an invalid input to the method and is referred to as *meaningless* [3]. For example, a test case  $\langle \langle \text{new Account}(10) \rangle, 20 \rangle$  is meaningless for the `withdraw` method.

We use OCL constraints as test oracles. A post-state assertion such as a method postcondition and a class invariant is used as a test oracle. The method under test is executed

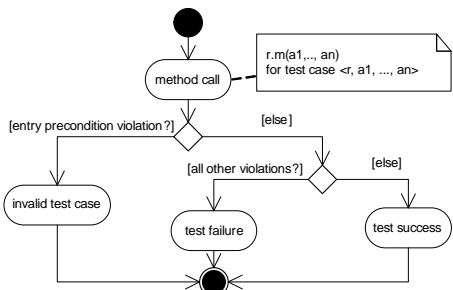


Figure 4. OCL constraints as test oracles

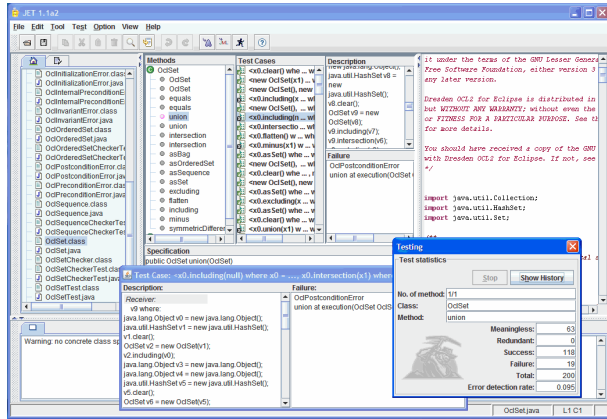


Figure 5. JET screenshot

with the generated test case. If the execution results in a post-condition violation error or a post-state invariant violation error, it is interpreted as a test failure because it means that the method doesn't satisfy its specification for that particular test data; there is an inconsistency between the code and its specification. Otherwise, it is interpreted as a test success because no code-specification inconsistency is detected.

## 4 Evaluation

We evaluated the feasibility and effectiveness of our approach through case studies and experiments. For the evaluation, we used our assertion-based, random testing tool named JET [4] that automates Java program testing (see Section 5 for a description of JET). Figure 5 shows a sample screenshot of JET. We first extended the JET tool to make it recognize OCL constraint violations and use them as decision procedures for checking validity of randomly generated test data and for deciding test results (see Sections 3.2 and 3.4).

We next selected two open-source Java applications that have formal UML models including class diagrams and OCL constraints. The first is the collection types defined in the OCL standard [11]; OCL supports a wide range of collection operations for writing constraints on collections. The second is an example application borrowed from the OCL

textbook by Warmer and Kleppe [14]. The example application is for a fictional company called Royal and Loyal (R&L) and manages loyalty programs for companies that offer their customers various kinds of bonuses. For both applications we used the Java implementations found in the Dresden OCL Toolkit distribution [7] (see Section 5 for a discussion of the Dresden tool). The OCL collection types consist of five Java classes with 1787 lines of source code (SLOC) and 336 lines of OCL constraints, and the R&L application consists of 19 classes with 1120 SLOC and 156 lines of OCL constraints. We manually translated the OCL constraints of both applications to AspectJ code, resulting in 698 SLOC and 541 SLOC, respectively.

We then ran the JET tool on each class of the applications, and it revealed several errors and deficiencies in both the Java implementations and their OCL constraints. In fact, for the OCL collection types, JET detected all the errors that we found earlier with a manual, JUnit-based testing. For example, the implementation of the `Set::union` operation was wrong; it returned only the argument set. This and similar programming errors were detected by the JET tool as inconsistencies between the OCL constraints and the Java implementations. It should be noted that several test failures were due to deficiencies in OCL specifications themselves. For example, collection operations such as `first` and `last` of types `Sequence` and `OrderedSet` are partial in that they are defined only for non-empty sequences and ordered sets. However, they are specified to be total functions in the standard [11]; they have no preconditions. Several other collection operations have a similar problem of missing or loose preconditions, which were detected as errors during test executions; e.g., attempting to access the first element of an empty sequence resulted in an exception. Finally, our AspectJ-based approach was also able to detect structural inconsistencies, as well. For example, when we translated and compiled the OCL constraint for the operation `Set::union(bag: Bag(T)): Bag(T)`, we received a compilation warning saying that our advice for the constraint has not been applied. We shortly learned that this warning was caused because there was no such method in the implementation; the return type of the corresponding Java method was `OclSet`, not `OclBag`.

We also performed a mutation testing experiment using the R&L application. We manually seeded total 30 faults to mutate the program. The JET tool was able to detect or kill 22 seeded faults or mutants, giving a 73% detection rate. Upon examining the undetected faults closely, we learned that there were interferences among the seeded faults. Once the interferences were removed, JET was able to detect most of the seeded faults (27 out of 30), giving a 90% detection rate. The remaining three faults were similar to what are referred to as equivalent mutants, mutants that have the same behavior as the original program. In our case, these were mutants that did not adversely affect the OCL constraints. In other words,

OCL constraints were not strong or detailed enough to distinguish these mutants from the original program.

## 5 Related Work

The basis of our work is automated random testing and the use of executable assertions as test oracles. Below we review some of most related work in these areas. JCrasher is a random testing tool to test the robustness of Java classes [6]. It generates a sequence of method calls randomly to cause the class under test to crash by throwing an uncaught exception. Our earlier work on JET also explored automated random testing but to detect inconsistencies between Java classes and their JML specifications [4]; JML is a formal behavioral interface specification language for Java. One difference between these two approaches is that JCrasher tests the whole class as a single unit to find an invalid method call sequence that results in an uncaught exception. JET, on the other hand, tests one method at a time by finding feasible call sequences and using JML assertions as decision procedures for feasibility and test results. Our current work extended JET to also recognize various OCL constraint violations.

One novel feature of JET is using formal specifications in constructing valid test data. The origin of this idea can be traced back to the use of formal specification as test oracles [9]. Peters and Parnas proposed a tool that generates a test oracle from formal program documentation written in tabular expressions [12]. The test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. Cheon and Leavens employed a JML runtime assertion checker as a test oracle engine to test Java programs [3]. In this paper we applied this idea to OCL and promoted it further by dynamically generating valid test data and feeding it to the oracle to detect inconsistencies between code and its OCL specification. Coppit and Haddox-Schatz performed case studies to evaluate the effectiveness of specification-based assertions by manually translating Object-Z specifications to executable assertions, and they concluded that such assertions can effectively reveal faults, as long as they adversely affect the program state [5].

There are several different approaches to check design constraints such as OCL constraints against an implementation [8]. A recent trend, however, is to use AOP languages such as AspectJ as constraints instrumentation languages by treating constraint checking as a crosscutting concern and handling it in a modular way [1, 13]. The constraint checking code exists separately from the implementation code, and the implementation is oblivious of the constraint checking. Demuth and Wilke, for example, developed an OCL verification tool called the Dresden OCL Toolkit that can not only interpret OCL constraints on a UML model and its Java implementation but also generate runtime constraints checking code written in AspectJ [7]. The Dresden Toolkit may be

adapted for use as a front-end tool for our proposed testing approach.

## 6 Conclusion

The main contribution of this paper is engineering an approach for automating Java program testing by combining existing two approaches: (1) translating OCL constraints into runtime checks written in AspectJ and (2) dynamic random testing based on executable assertions. Our case studies and experiments, though limited in their scopes, show the effectiveness of our approach in detecting errors in both implementations and OCL constraints. Thus, our approach also provides a practical means for debugging OCL constraints themselves. We, therefore, expect our approach to promote a practical use of OCL in design and in programming, as well.

## Acknowledgment

The work of the authors was supported in part by NSF grants CNS-0509299 and CNS-0707874.

## References

- [1] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, September 25-30, 2005*, pages 687–690, Sept. 2005.
- [2] Y. Cheon, C. Avila, S. Roach, C. Munoz, N. Estrada, V. Fierro, and J. Romo. An aspect-based approach to checking design constraints at run-time. In *ITNG 2009: 6th International Conference on Information Technology: New Generations, April 27-29, 2009, Las Vegas, NV*, pages 223–228. IEEE Computer Society, 2009.
- [3] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, volume 2374 of *LNCS*, pages 231–255. Springer-Verlag, June 2002.
- [4] Y. Cheon and C. E. Rubio-Medrano. Random test data generation for Java classes annotated with JML specifications. In *SERP, Volume II, June 25–28, 2007, Las Vegas, Nevada*, pages 385–392, June 2007.
- [5] 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.
- [6] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software—Practice and Experience*, 34(11):1025–1050, Sept. 2004.
- [7] B. Demuth and C. Wilke. Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, July 25-31, Ufa, Russia, 2009*, pages 687–690, 2009.

- [8] L. Frohofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 313–322. IEEE Computer Society, 2007.
- [9] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Prog. Lang. Syst.*, 3(3):211–223, 1981.
- [10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, Berlin, June 2001.
- [11] OMG. *UML 2.0 OCL Specification*. Object Management Group, Oct. 2003. Available from <http://www.omg.org/docs/ptc/03-10-14.pdf> (retrieved on Oct. 15, 2008).
- [12] 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.
- [13] M. Richters and M. Gogolla. Aspect-oriented monitoring of UML and OCL constraints. In *The 4th AOSD Modeling with UML Workshop, San Francisco, CA, October 20, 2003*, 2008. Co-located with UML 2003.
- [14] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, second edition, 2003.