IMAG

Institut d'Informatique et de
Mathématiques Appliquées
de Grenoble

LSR

Laboratoire Logiciels, Systèmes, Réseaux

RAPPORT DE RECHERCHE

Jartege: a Tool for Random Generation of Unit Tests for Java Classes

Catherine Oriat

RR 1069 **Juin 2004**

B.P. 72 – 38402 SAINT MARTIN D'HERES CEDEX – France

Centre National de la Recherche Scientifique
Institut National Polytechnique de Grenoble
Université Joseph Fourier Grenoble I

Jartege: a Tool for Random Generation of Unit Tests for Java Classes

Catherine Oriat
LSR-IMAG, Grenoble
email: Catherine.Oriat@imag.fr

Résumé

Ce rapport présente Jartege, un outil qui permet la génération aléatoire de tests unitaires pour des classes Java spécifiées en JML. JML (Java Modeling Language) est un langage de spécification pour Java qui permet d'écrire des invariants pour des classes, ainsi que des pré- et des post-conditions pour des opérations. Comme dans l'outil JML-JUnit, nous utilisons les spécifications JML d'une part pour éliminer des cas de test non pertinents, et d'autre part comme oracle de test. Jartege génère de façon aléatoire des cas de test, qui consistent en une séquence d'appels de constructeurs et de méthodes des classes sous test. L'aspect aléatoire de l'outil peut être paramétré en associant des poids aux classes et aux opérations, et en contrôlant le nombre d'instances créées pour chaque classe sous test. L'utilisation pratique de Jartege est illustrée par une petite étude de cas.

Mots-clés

Test, test unitaire, génération aléatoire de cas de test, Java, JML

Abstract

This report presents Jartege, a tool which allows random generation of unit tests for Java classes specified in JML. JML (Java Modeling Language) is a specification language for Java which allows one to write invariants for classes, and pre- and postconditions for operations. As in the JML-JUnit tool, we use JML specifications on the one hand to eliminate irrelevant test cases, and on the other hand as a test oracle. Jartege randomly generates test cases, which consist of a sequence of constructor and method calls for the classes under test. The random aspect of the tool can be parameterized by associating weights to classes and operations, and by controlling the number of instances which are created for each class under test. The practical use of Jartege is illustrated by a small case study.

Keywords

Testing, unit testing, random generation of test cases, Java, JML

Jartege: a Tool for Random Generation of Unit Tests for Java Classes

Catherine Oriat
LSR-IMAG, Grenoble
email: Catherine.Oriat@imag.fr

1 Introduction

Main validation technique in software engineering, program testing aims at ensuring that the program is *correct*, i.e. conforms to its specifications. As the input domain of a program is usually very large or infinite, exhaustive testing, which consists in testing the program for all its possible inputs, is in general impossible. The objective of testing is thus rather to improve the software quality by finding faults in it. Testing is an important activity of software development, whose cost is usually estimated to about 40% of the total cost of software development, exceeding the cost of code writing.

A test campaign for a program requires several steps: design and development of test sets, execution and results examination (or *oracle*). Considering the cost of testing, it is interesting to automate some of these steps.

For Java programs, the JUnit framework [JUn, BG98] allows the developer to write an oracle for each test case, and to automatically execute test sets. JUnit in particular permits to automatically regression test several test sets.

If a formal specification is available, it can be translated into assertions which can be checked at runtime, and thus serve as a test oracle. For instance, the DAISTS system [GMH81] compiles algebraic axioms of an abstract data type into consistency checks; Rosenblum's APP pre-processor allows the programmer to write assertions for C programs [Ros92]; the Eiffel *design and contract* approach integrates assertions in the programming language [Mey88, Mey92].

It is also interesting to automate the *development* of tests. We can distinguish between two groups of strategies to produce test sets: random and systematic strategies. Systematic strategies, such as functional testing or structural testing, consist in decomposing the input domain of the program in several subdomains, often called "partitions".

Many systematic strategies propose to derive test cases from a formal specification [Gau95]. For instance, the Dick and Faivre method [DF93], which consists in constructing a finite state automaton from the formal specification and in selecting test cases as paths in this automaton, has been used as a basis by other approaches, in particular Casting [ABM97] or BZTT [LPU02]. BZTT uses B or Z specifications to generate test cases which consist in placing the system in a boundary state and calling an operation with a boundary value.

In contrast to systematic methods, random testing generally does not use the program nor the specification to produce test sets. It may use an *operational profile* of the program, which describe how the program is expected to be used. The utility of random testing is controversial in the testing community. It is usually presented as the poorest approach for selecting test data [Mye94]. However, random testing has a few advantages which make us think that it could be a good complement to systematic testing:

- random testing is cheap and rather easy to implement. In particular, it can produce large or very large test sets;

- it can detect a substantial number of errors at a low cost [DN84, HT90, Nta01].

Moreover, if an operational profile of the program is available,

- random testing allows early detection of the failures that are most likely to appear when using the program [FHLS98];
- it can be used to evaluate the program *reliability* [HT90, Ham94].

However, partition testing can be much more effective at finding failures, especially when the strategy defines some very small subdomains with a high probability to cause failures [WJ91].

Among the best practices of *extreme programming*, or XP [Bec99, Bec00], are *continuous testing* and *code refactoring*. While they are writing code, developers should write corresponding unit tests, using a testing framework such as JUnit. Unit testing is often presented as a support to refactoring: it gives the developer confidence that the changes have not introduced new errors. However, code refactoring often requires to change some of the corresponding unit tests as well. If a class has to be tested intensively, the amount of code corresponding to the tests often exceeds the amount of code of the class. Both practices can therefore be hard to conciliate.

This report proposes to use random generation of tests to facilitate countinuous testing and code refactoring in the context of extreme programming. We presents Jartege, a tool for random generation of unit tests for Java classes specified in JML, which aims at easily producing numerous test cases, in order to detect a substantial number of errors at a low cost.

The rest of the report is organized as follows. Section 2 introduces the approach. Section 3 presents a case study which consists in modeling bank accounts. This case study will serve to illustrate the use of our testing tool. Section 4 presents our tool Jartege and how it can be used to test the bank account case study. Section 5 introduces more advanced features of Jartege, which allow one to parameterize its random aspect. Section 6 shows the errors which are detected by test cases generated by Jartege in the case study. Section 7 presents and compares related approaches. Section 8 discusses some points about random generation of tests and draw future work we intend to undertake around Jartege.

2 Approach

JML (Java Modeling Language) is a specification language for Java inspired by Eiffel, VDM and Larch, which was designed by Gary Leavens and his colleagues [JML, LBR03, LPC+03]. Several teams are currently still working on JML design and tools around JML [BCC+03]. JML allows one to specify various assertions in particular invariants for classes as well as pre- and postconditions for methods.

The JML compiler (jmlc) [CL02a] translates JML specifications into assertions checked at runtime. If an assertion is violated, then a specific exception is raised. In the context of a given method call, the JML compiler makes a useful difference between an *entry precondition* which is a precondition of the given method, and an *internal precondition*, which is a precondition of an operation being called, at some level, by the given method.

The JML-JUnit tool [CL02b] generates JUnit test cases for a Java program specified in JML, using the JML compiler to translate JML specifications into test oracles. Test cases are produced from a test fixture and parameter values supplied by the user.

Our approach is inspired by the JML-JUnit tool: we propose to generate random tests for Java programs specified in JML, using this specification as a test oracle, in the JML-JUnit way. Our aim is to produce a big number of tests at a low cost, in order to facilitate unit testing.

To implement these ideas, we started developing a prototype tool, called *Jartege* for *Java Random Test Generator*. Jartege is designed to generate *unit tests* for Java classes specified with JML. By unit tests, we here mean tests for some operations in a single class or a small cluster of classes. In our context, a *test case* is a Java method which consists of *a sequence of operation (constructor or method) calls*.

As in the JML-JUnit tool, we use the JML specification to assist test generation in two ways:

1. It permits the rejection of test sequences which contain an operation call which violates the operation *entry precondition*. We consider that these test sequences are not interesting because they detect errors which correspond to a fault *in the test program*.

   (Although such sequences could be used to detect cases when a precondition is too strong, this goal would prevent us from producing long sequences of calls. Thus, we choose to trust the specification rather than the code. We can note that if an operation uses a method whose precondition is too strong, this will produce an *internal precondition* error and the sequence will *not* be rejected.)

2. The specification is also used as a test oracle: the test detects an error when another assertion (e.g. an invariant, a postcondition or an *internal precondition*) is violated. Such an error corresponds to a fault in the Java program or in the JML specification.

Our work has been influenced by the Lutess tool [Par96, dBORZ99], which aims at deriving test data for synchronous programs, with various generation methods, in particular a purely random generation and a generation guided by operational profiles. The good results obtained by Lutess, which won the best tool award of the first feature interaction detection contest [dBZ99], have encouraged us to consider random generation of tests as a viable approach.

3 Case Study

In this section, we present a case study to illustrate the use of Jartege. This case study defines *bank accounts* and some operations on these accounts.

3.1 Informal Specification of the Bank Accounts

We describe here an informal specification of bank accounts:

1. An account contains a certain available amount of money (its *balance*), and is associated with a minimum amount that this account may contain (the *minimum balance*).

2. It is possible to credit or debit an account. A debit operation is only possible if there is enough money on the account.

3. One or several last credit or debit operations may be cancelled.

4. The minimum balance of an account may be changed.

3.2 Bank Account Modeling

To represent accounts, we define a class Account with two attributes: balance and min which respectively represent the balance and the minimum balance of the account, and three methods: credit, debit and cancel. In order to implement the cancel operation, we associate with each account an *history*, which is a linked list of the previous balances of the account. The class History has one attribute, balance, which represents the balance of its associated account before the last credit or debit operation. With each history is associated its preceding history. Figure 1 shows the UML class diagram of bank accounts.

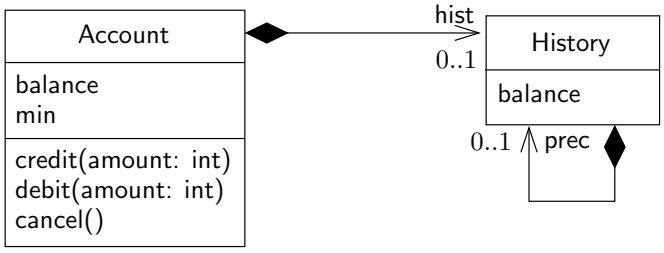

Figure 1: UML diagram of the case study

3.3 JML Specification and Java Implementation

We implement both class Account and History in Java, and specify them with JML. We choose to write *lightweight public specifications* to emphasize that these specifications are destined for clients and need not be complete.

In order to forbid uncontrolled modifications of the attributes, we declare them as private and define associated access methods: getBalance, getMin and getHist in class Account and getBalance and getPrec in class History. These methods are specified as *pure* methods in JML because they are side effect free, which allows us to use them in JML assertions.

The class Account has an invariant which specifies that the balance of an account must always be greater than the minimum balance.

```
/* Class of bank accounts. */
public class Account {
    /* Invariant of class Account. */
    /*@ public invariant getBalance( ) >= getMin( ); */

    private int balance; // The balance of this account
    private int min;     // The minimum balance
    private History hist; // The history list of this account

    /* The balance of this account. */
    public /*@ pure */ int getBalance( ) { return balance; }
```

```
/* The history list of this account. */
public /*@ pure */ History getHist() { return hist; }

/* The minimum balance of this account. */
public /*@ pure */ int getMin() { return min; }
```

The constructor of class Account constructs an account with the specified balance and the specified minimum balance. Its precondition asserts that the specified balance is greater than the specified minimum balance.

```
/* Constructs an account with the specified balance and
 * minimum balance. */
/*@ requires balance >= min; */
public Account(int balance, int min) {
    this.balance = balance;
    this.min = min;
    this.hist = null;
}
```

As the minimum balance min is a private attribute, we have to introduce a method to change its value. The method setMin(*int* min) sets the minimum balance to the specified value. Its precondition asserts that the balance is greater than the specified minimum value.

```
/* Sets the minimum balance to the specified value. */
/*@ requires getBalance() >= min; */
public void setMin(int min) { this.min = min; }
```

The method credit(*int* amount) credits the account with the specified amount. Its precondition requires the amount to be positive. Its postcondition asserts that the new balance is the former balance augmented by the specified amount, that a new history is created with balance the former balance of the account and with previous history the former history of the account. Its exceptional postcondition asserts that the method should never terminate abruptly.

```
/* Credits this account with the specified amount. */
/*@ requires amount >= 0;
 *@ ensures getBalance() == \old(getBalance()) + amount &&
 *@         \fresh(getHist()) &&
 *@         getHist().getBalance() == \old(getBalance()) &&
 *@         getHist().getPrec() == \old(getHist());
 *@ signals (Exception e) false;
 */
public void credit(int amount) {
   hist = new History(balance, getHist());
   balance = balance + amount;
}
```

The debit operation, which is very similar to the credit operation, is not detailed here. It has the additional precondition that the balance decreased by the specified amount is greater than the minimum balance.

The method cancel cancels the last credit or debit operation. Its precondition requires that the history is not null, which means that at least one operation of credit or debit has taken place since the account was created. Its postcondition ensures that the balance and the history of the account have been updated with their former values.

```
  /* Cancels the last credit or debit operation. */
  /*@ requires getHist() != null;
   *@ ensures getHist() == \old(getHist().getPrec()) &&
   *@         getBalance() == \old(getHist().getBalance());
   *@ signals (Exception e) false;
   */
  public void cancel() {
      balance = hist.getBalance();
      hist = hist.getPrec();
  }
} // End of class Account
```

We do not define any JML assertion for the class History.

```
/* Class of histories. */
public class History {
  private int balance;  // The balance of this history.
  private History prec; // The preceding history.

  /* Constructs a history with the specified balance and preceding history. */
  public History(int balance, History prec) {
      this.balance = balance; this.prec = prec;
  }

  /* The balance of this history. */
  public /*@ pure */ int getBalance() { return balance; }
  /* The preceding history. */
  public /*@ pure */ History getPrec() { return prec; }
} // End of class History
```

4 Jartege

Jartege (Java Random Test Generator) is a framework for automatic random generation of unit tests for Java classes specified with JML. This approach consists in producing test programs which are composed of test cases, each test case consisting of randomly chosen sequences of method calls for each class under test. Each generated test program can be executed to test the classes, and re-executed later on either after having corrected some faults or for regression test.

The tool is designed to produce *unit tests*, i.e. tests composed of calls of some methods which belong to a few classes. As noticed in [LTFWD00], because of complex dependences that exist between classes in object-oriented programs, it is usually not possible to test a method or a class in complete isolation. Jartege thus is able to generate test cases which allow the integration of several classes.

4.1 Practical Use of Jartege

Suppose we wish to generate tests for the classes Account and History. We write the following Java program:

```
import jartege.*;
/** Jartege test cases generator for classes Account and History. */
class TestGen {
  public static void main(String[] args) {
    // Creates a class tester
    ClassTester t = new ClassTester();
    // Adds the specified classes to the set of classes under test
    t.addClass("Account");
    t.addClass("History");
    // Generates a test class TestBank, made of 100 test cases.
    // For each test case, the tool tries to generate 50 method calls.
    t.generate("TestBank", 100, 50);
  }
}
```

The main class of the Jartege framework is ClassTester. This class must be instantiated to allow the creation of test programs.

The method addClass(String className) adds the class className to the set of classes under test. In this example, we wish to generate tests for the classes Account and History.

The method generate(String className, *int* numberOfTests, *int* numberOfMethodCalls) generates a file *className*.java which contains a class called *className*. This class is composed of *numberOfTests* test cases. For each test case, the tool makes *numberOfMethodCalls* attempts to generate a method call of one of the classes under test. Using the accessible constructors, the tool constructs objects which serve as parameters for these method calls.

When the program TestGen is executed, it produces a file TestGen.java which contains a main program. This main program calls successively 100 test methods test1(), test2() ... test100(). Each test method contains about 50 method calls.

While the program is generated, Jartege executes *on the fly* each operation call, which allows it to eliminate calls which violate the operation precondition. When this precondition is strong, it may happen that the tool does not succeed in generating a call for a given method, which explains that *about* 50 method calls are generated.

4.2 Test Programs Produced by Jartege

A test program produced by Jartege is a class with a main method which consists in calling sequentially all generated test cases. Each test case consists of a sequence of constructor and method calls of the classes under test. Here is an example of such a test case:

```
// Test case number 1
public void test1() throws Exception {
  try {
    Account ob1 = new Account(1023296578, 223978640);
    ob1.debit(152022897);
```

```
      History ob2 = new History(1661966075, (History) null);
      History ob3 = new History(-350589348, ob2);
      History ob4 = ob2.getPrec();
      int ob5 = ob3.getBalance();
      ob1.cancel();
      // ...
    } catch (Throwable _except) {
      error(_except, 1);
    }
}
```

For each test method, if a JML exception is raised, then an error message (coming from jmlc) is printed. The test program terminates by printing an assessment of the test. As an example, here is an excerpt of what is printed with a generated program TestBank.java:

```
1) Error detected in class TestBank by method test2:
   org.jmlspecs.jmlrac.runtime.JMLInvariantError:
   By method "credit@post⟨Account.java:79:18⟩" of class "Account"
   for assertions specified at Account.java:11:32 [...]
   at TestBank.test2(TestBank.java:138)
[...]
Number of tests: 100
Number of errors: 71
Number of inconclusive tests: 0
```

The program has detected 71 errors. The first error detected comes from a violation of the invariant of class Account (specified line 11), which happened after a credit operation.

The test program also indicates the number of *inconclusive* tests. A test case is inconclusive when it does not allow one to conclude whether the program behaviour is correct or not. A test program generated by Jartege indicates that a test case is inconclusive when it contains an operation call whose *entry precondition* is violated. As Jartege is designed to eliminate such operation calls, this situation may only arise when the code or the specification of one of the classes under test has been modified after the test file was generated. A high number of inconclusive tests indicates that the test file is no longer relevant.

5 Controlling Random Generation

If we leave everything to chance, Jartege might not produce interesting sequences of calls. Jartege thus provides a few possibilities to parameterize its random aspect. These features can be useful for stress testing, for instance if we want to test more intensively a given method. More generally, they allow us to define an *operational profile* for the classes under test, which describe how these classes are likely to be used by other components.

5.1 Weights

With each class and operation of a class is associated a weight, which defines the probability that a class will be chosen, and that an operation of this class will be called. In particular, it is possible to forbid to call some operation by associating a null weight with it. By

default, all weights are equal to 1. A weight can be modified by a weight change method. In particular:

- changeAllMethodsWeight (String className, double weight) changes the weight of all methods in the class className to the specified weight.
- changeMethodWeight (String className, String methodName, double weight) changes the weight of the specified method(s) in the class className to the specified weight.
- changeMethodWeight (String className, String methodName, String [] signature, double weight) changes the weight of the method by its name and its signature to the specified weight.

5.2 Creation of Objects

Objects creation is commanded by *creation probability functions*, which define the probability of creating a new object according to the number of existing objects of the class against that of reusing an already created object. If this probability is low, Jartege is more likely to reuse an already created object than to construct a new one. This allows the user either to create a predefined number of instances for a given class, or on the opposite, to create numerous instances for a class.

In the example of bank accounts, it is not very interesting to create many accounts. It is possible to test the class Account more efficiently for example by creating a unique account and by applying numerous method calls to it.

The function changeCreationProbability(String className, CreationProbability creationProbabilityFunction) changes the creation probability function associated with the specified class to the the specified creation probability function.

The interface CreationProbability contains a unique method

double theFunction(int nbCreatedObjects)

which must satisfy the condition

$$\text{theFunction}(0) = 1;$$
$$\text{theFunction}(n) \in [0, 1], \forall n \geq 1.$$

The class ThresholdProbability allows one to define *threshold probability functions* whose value is 1 under some threshold s and 0 above.

$$\text{theFunction}(n) = 1, \text{ if } n < s;$$
$$\text{theFunction}(n) = 0, \text{ otherwise.}$$

A threshold probability function with threshold s allows one to define at most s instances of a given class. We can for instance forbid the creation of more than one instance of Account by adding the following statement in the test generator:

```
t.changeCreationProbability("Account", new ThresholdProbability(1));
```

5.3 Parameter Generation of Primitive Types

When a method has a strong precondition, the probability that Jartege, without any further indication, will generate a call to this method which does not violate this precondition is low. For primitive types, Jartege provides the possibility to define generators for some parameters of a given method.

For example, the precondition of the debit operation requires the parameter amount to be in range

$$[0, \text{getBalance}() - \text{getMin}()].$$

Let us suppose the range is small, in other words that the balance is closed to the minimum balance. If Jartege chooses an amount to be debited entirely randomly, this amount is not likely to satisfy the method precondition.

Jartege provides a way of generating parameter values of primitive types for operations. For this, we define a class JRT_Account as follows.

```
import jartege.RandomValue;
public class JRT_Account {
  private Account theAccount; // The current account

  /* Constructor. */
  public JRT_Account(Account theAccount) { this.theAccount = theAccount; }

  /** Generator for the first parameter of operation debit (int). */
  public int JRT_debit_int_1() {
     return RandomValue.intValue(0,
        theAccount.getBalance() - theAccount.getMin());
  }
}
```

The class JRT_Account must contain a private field of type Account which will contain the current object on which an operation of class Account is applied. A constructor allows Jartege to initialize this private field. The class also contains one parameter generation method for each parameter for which we specify the generation of values. In the example, to specify the generation of the first parameter of operation debit(*int* amount), we define the method *int* JRT_debit_int_1(). We use the signature of the operation in the name of the method to allow overloading. The method RandomValue.intValue(*int* min, *int* max) chooses a random integer in range [min, max].

5.4 Fixtures

If we want to generate several test cases which operate on a particular set of objects, we can write a *test fixture*, in a similar way to JUnit. A test fixture is a class which contains:

- attributes corresponding to the objects a test operates on;
- an optional setUp method which defines the preamble of a test case (which typically constructs these objects);
- an optional tearDown method which defines the postamble of a test case.

6 Applying Jartege to the Case Study

The 100 test cases generated by Jartege, showing 71 failures, revealed three different errors: one error caused by a credit operation, and two errors caused by a cancel operation. We extracted the shorter sequence of calls which resulted in each failure and obtained the following results. We also changed the parameter values and added some comments for more readability.

Error 1. The credit operation can produce a balance inferior to the previous balance because of an integer overflow.

```
public void test1() {
  Account ob1 = new Account(250000000, 0);
  ob1.credit(2000000000); // Produces a negative balance,
}                         // below the minimum balance.
```

Error 2. The cancel operation can produce an incorrect result if it is preceded by a setMin operation which changes the minimum balance of the account to a value which is superior to the balance before cancellation.

```
public void test11() {
  Account ob1 = new Account(-50, -100);
  ob1.credit(100);
  ob1.setMin(0);
  ob1.cancel(); // Restores the balance to a value
}               // inferior to the minimum balance.
```

Error 3. The third error detected is a combination of an overflow on a debit operation (similar to Error 1, which comes from an overflow on a credit operation) and of the second error.

```
public void test13() {
  Account ob1 = new Account(-1500000000, -2000000000);
  ob1.debit(800000000); // Produces a positive balance.
  ob1.setMin(0);
  ob1.cancel();  // Restores the balance to a value
}                // inferior to the minimum balance.
```

We have the feeling that the three errors detected with test cases generated by Jartege are not totally obvious, and could have easily been forgotten in a manually developed test suite. Errors 2 and 3 in particular require three method calls to be executed in a specific order and with particular parameter values.

It must be noted that the case study was originally written to show the use of JML to undergraduate students, without us being aware of the faults.

7 Comparison with Related Work

Our work has been widely inspired by the JML-JUnit approach [CL02b]. The JML-JUnit tool generates test cases for a method which consist of a combination of calls of this method

with various parameter values. The tester must supply the object invoking the method and the parameter values. With this approach, interesting values could easily be forgotten by the tester. Moreover, as a test case only consists of one method call, it is not possible to detect errors which result of several calls of different methods. At last, the JML-JUnit approach compels the user to construct the test data, which may require the call of several constructors. Our approach thus has the advantage of being more automatic, and of being able to detect more potential errors.

Korat [BKM02] is a tool also based on the JML-JUnit approach, which allows exhaustive testing of a method for all objects of a bounded size. The tools automatically construct all non isomorphic test cases and execute the method on each test case. Korat therefore has the advantage over JML-JUnit of being able to construct the objects which invoke the method under test. However, test cases constructed by Korat only consist of one object construction and one method invocation on this object.

Tobias [Led02, MLBdB02] is a combinatorial testing tool for automatic generation of test cases derived from a "test pattern", which abstractly describes a test case. Tobias was first designed to produce test objectives for the TGV tool [JM99] and was then adapted to produce test cases for Java programs specified in JML and for programs specified in VDM. The main problem of Tobias is the combinatorial explosion which happens if one tries to generate test cases which consist of more than a couple of method calls. Jartege was designed to allow the generation of long test sequences without facing the problem of combinatorial explosion.

8 Discussion and Future Work

Jartege is only in its infancy and a lot of work remains to be done.

Primitive values generation for methods parameters is currently done manually by writing primitive parameters generating methods. Code for these methods could be automatically constructed from the JML precondition of the method. This could consist in extracting range constraints from the method precondition and automatically produce a method which could generate meaningful values for the primitive parameters.

Jartege easily constructs test cases which consist of hundreds of constructors and methods calls. It would be useful to develop a tool for extracting a minimum sequence of calls which results in a given failure.

We developed Jartege in Java, and we specified some of its classes with JML. We applied Jartege to these classes to produce test cases, which allowed us to experiment our tool on a larger case study and to detect a few errors. We found much easier to produce tests with Jartege than to write unit tests with JUnit or JML-JUnit. We intend to continue our work of specifying Jartege in JML and testing its classes with itself. We hope that this real case study will help us to evaluate the effectiveness and scalability of the approach.

A comparison of our work with other testing strategies still remains to be done. We can expect systematic methods, using for instance boundary testing such as BZTT [LPU02], to be able to produce more interesting test cases that ours. Our goal is certainly not to pretend that tests produced randomly can replace tests produced by more sophisticated methods, nor a carefully designed test set written by an experienced tester.

Our first goal in developing Jartege was to help the developer to write unit tests for unstable Java classes, thus for "debug unit testing". It would also be interesting to use Jartege to evaluate the reliability of a stable component before it is released. Jartege

provides some features to define an operational profile of a component, which should allow statistical testing. However, the definition of a correct operational profile, especially in the context of object-oriented programming, is a difficult task. Moreover, the relation between test sets generated by Jartege and the reliability of a component requires more theoretical work, one difficult point being to take into account the state of the component.

9 Conclusion

This report presents Jartege, a tool for random generation of unit tests for Java classes specified in JML. The aim of the tool is to easily produce numerous test cases, in order to detect a substantial number of errors without too much effort. It is designed to produce automated tests, which can in part replace tests written by the developer using for instance JUnit. We think that the automatic generation of such unit tests should facilitate continuous testing as well as code refactoring in the context of extreme programming.

The JML specifications are used on the one hand to eliminate irrelevant test cases, and on the other hand as a test oracle. We think that the additional cost of specification writing should be compensated by the automatic oracle provided by the JML compiler, as long as we wish to intensively test the classes. Moreover, this approach has the advantage of supporting the debugging of a specification along with the corresponding program. This allows the developer to increase his confidence in the specification and to use this specification in other tools.

Most test generation methods are deterministic, while our approach is statistical. We do not wish to oppose both approaches, thinking that they both have their advantages and drawbacks, and that a combination of both could be fruitful.

At last, we found JML to be good language to start learning formal methods. Its Java-based syntax makes it easy to learn for Java programmers. As JML specifications are included in Java source code as comments, it is easy to develop and debug a Java program along with its specification. Moreover, automatic test oracles as well as automatic generation of test cases are good reasons of using specification languages such as JML.

References

[ABM97] Lionel Van Aertryck, Marc Benveniste, and Daniel Le Métayer. Casting: a formally based software test generation method. In *Proceedings of the First IEEE Internatinal Conference on Formal Engineering Methods — ICFEM'97, Hiroshima, Japan*, pages 101–111, November 1997.

[BCC+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Department of Computer Science, University of Nijmegen, March 2003.

[Bec99] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.

[Bec00] Kent Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Reports*, 3(7):51–56, 1998.

[BKM02] Chandrasekhar Boyapati, Safraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis — ISSTA'02, Rome*, pages 123–133, July 2002.

[CL02a] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Hamid R. Arabnia and Youngsong Mun (eds.), International Conference on Software Engineering Research and Practice — SERP'02, Las Vegas, Nevada*, pages 322–328. CSREA Press, 2002.

[CL02b] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Boris Magnusson (ed.), 16th European Conference on Object-Oriented Programming — ECOOP'02, Malaga, Spain*, number 2374 in Lecture Notes in Computer Science, pages 231–255. Springer-Verlag, June 2002.

[dBORZ99] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *International Conference on Software Engineering — ICSE'99, Los Angeles, USA*. ACM Press, May 1999.

[dBZ99] Lydie du Bousquet and Nicolas Zuanon. An overview of Lutess: a specification-based tool for testing synchronous software. In *14th IEEE International Conference on Automated Software Engineering — ASE'99*, pages 208–215, October 1999.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of FME'93*, number 670 in LNCS, pages 268–284. Springer-Verlag, April 1993.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[FHLS98] Phyllis G. Frankl, Richard G. Hamlet, Bev LittleWood, and Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998.

[Gau95] Marie-Claude Gaudel. Testing can be formal too. In *Proceedings of TAPSOFT'95, Aarhus, Denmark*, number 915 in LNCS, pages 82–96. Springer-Verlag, May 1995.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–233, July 1981.

[Ham94] Richard Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[HT90] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12), December 1990.

[JM99] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In *Proceedings of the 11th International Conference on Computer Aided Verification — CAV'99, Trento, Italy*, number 1633 in LNCS, pages 108–121. Springer-Verlag, July 1999.

[JML] The Java Modeling Language (JML) Home Page, http://www.cs.iastate.edu/~leavens/JML.

[JUn] The JUnit Home Page, http://www.junit.org.

[LBR03] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java.* Department of Computer Science, Iowa State University, 1998–2003.

[Led02] Yves Ledru. The TOBIAS test generator and its adaptation to some ASE challenges (position paper). In *Workshop on the State of the Art in Automated Software Engineering, ICS Technical Report UCI-ICS-02-17, University of California, Irvine, USA*, 2002.

[LPC+03] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. *JML Reference Manual.* (Draft), April 2003.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In *Proceedings of FME'02, Formal Methods Europe, Copenhaguen, Denmark*, number 2391 in LNCS, pages 21–40. Springer-Verlag, July 2002.

[LTFWD00] Yvan Labiche, Pascale Thévenod-Fosse, Hélène Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. In *Proceedings of the 22nd International Conference on Software Engineering — ICSE'00, Limerick, Ireland*, pages 136–145. ACM, June 2000.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey92] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[MLBdB02] Olivier Maury, Yves Ledru, Pierre Bontron, and Lydie du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *Third VDM Workshop (at FME'02), Copenhaguen, Denmark*, July 2002.

[Mye94] G. J. Myers. *The Art of Software Testing.* John Wiley and Sons, New York, 1994.

[Nta01] Simeon C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, July 2001.

[Par96] Ioannis Parissis. *Test de logiciels synchrones spécifiés en Lustre.* PhD thesis, Grenoble, France, Septembre 1996.

[Ros92] David S. Rosenblum. Towards a method of programming with assertions. In *International Conference on Software Engineering — ICSE'92.* IEEE Computer Society Press, 1992.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.