Parameterized Unit Tests with Unit Meister

Nikolai Tillmann
nikolait@microsoft.com

Wolfram Schulte
schulte@microsoft.com

Microsoft Research
One Microsoft Way, Redmond WA USA

ABSTRACT

Parameterized unit tests extend the current industry practice of using closed unit tests defined as parameterless methods. Traditional closed unit tests are re-obtained by instantiating the parameterized unit tests. We have developed the prototype tool *Unit Meister*, which uses symbolic execution and constraint solving to automatically compute a minimal set of inputs that exercise a parameterized unit test given certain coverage criteria. In addition, the parameterized unit tests can be used as symbolic summaries during symbolic execution, which allows our approach to scale for arbitrary abstraction levels. Unit Meister has a command-line interface, and is also integrated into Visual Studio 2005 Team System.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications — *Methodologies*; D.2.5 [Software Engineering]: Testing and Debugging — *Testing tools*

General Terms: Design, Verification

Keywords: unit testing, algebraic data types, symbolic execution, automatic test input generation, constraint solving

1. INTRODUCTION

Object-oriented unit tests are written as test classes with test methods. A test method is a method without input parameters. It represents a single test case and typically executes a method of a class-under-test with fixed arguments and verifies that it returns the expected result.

Unit tests are a key component of software engineering. Being of such importance, many companies now provide tools, frameworks and services around unit tests. Tools range from specialized test frameworks, as for example integrated in Visual Studio Team System [15] (VSUnit), to automatic unit-test generation, e.g. as provided by Parasoft's JUnit Test Tool [18]. However these tools don't provide any guidance for:

- which tests should be written,
- how to come up with a minimal number of test cases and
- what guarantees the test cases provide.

Parameterized unit tests (PUTs) is a *new* methodology extending the current industry practice of closed unit tests (i.e. test methods without input parameters). Test methods are generalized by allowing parameters. This serves two purposes. First, parameterized test methods are *specifications* of the behavior of the methods-under-test: they do not only provide exemplary arguments to the methods-under-test, but ranges of such arguments. Second, parameterized unit tests describe a set of traditional unit tests which can be obtained by *instantiating* the parameterized test methods with given argument sets. Instantiations should be chosen so that they exercise different code paths of the methods-under-test.

We instrument parameterized unit tests using symbolic execution techniques. To this end, we execute a PUT symbolically, assigning symbolic variables to its parameters. Each symbolic execution path results in a *path condition*, and finding solutions to that condition results in instantiations of the parameters of the PUT. If the methods-under-test have only finitely many paths and if a PUT passes for the chosen instantiations, the PUT would pass for all possible instantiations; a result which goes back to [13]. For dealing with PUTs with an unbounded number of paths, we impose bounds on loops and recursion; even in that case, we can still obtain an unbiased set of test cases with high code coverage.

PUTs can also be used as summaries of the behavior of the methods specified in them. During symbolic execution, we can use these summaries of already tested methods instead of re-exploring them. This increases the performance of symbolic execution, since when testing a component using summaries of already tested classes, fewer paths must be investigated, and thus fewer test cases are generated while still maintaining the same coverage of the currently tested component.

Admittedly, writing open, parameterized unit tests is more challenging than writing closed, traditional unit tests. However, we believe that the benefit of automatic and comprehensive test case generation outweighs the additional effort.

We have developed the prototype tool Unit Meister, which uses symbolic execution of .NET assemblies in the Exploring Runtime, XRT[12], to instrument PUTs. Unit Meister's *contributions* are:

- It performs symbolic execution of object-oriented programs with symbolic references.
- It can use summaries obtained from PUTs for symbolic reasoning to avoid re-execution of summarized methods.
- The automated case analysis scales by using such summaries.
- It automatically generates concrete inputs to exercise explored code paths in many cases.
- Evaluations have shown Unit Meister's ease-of-use and usefulness [20].

We plan to integrate the methodology of PUTs into the forthcoming Visual Studio 2005 Team System product; Unit Meister is currently integrated in Visual Studio as an Add-In.

In Section 2 we give an overview of the methodology of PUTs, in Section 3 we describe how our prototype tool can be used, and in Section 4 we present a brief overview of related work.

An extended version of the overview and the related work as well as a short presentation of Unit Meister's axiom analysis framework and its evaluation on parts of the .NET base class library can be found in [20].

2. OVERVIEW

2.1 Traditional Unit Tests

Using the conventions of NUnit[16, 17] and VSUnit[15], we define unit tests as test methods contained in test classes. A parameterless method decorated with a custom attribute like `[TestMethod]` is a test method. Usually, each unit test explores a particular aspect of the behavior of the class-under-test.

Here is a unit test method written in C# for VSUnit that adds an element to a .NET `ArrayList` instance. The test first creates a new array list, where the parameter to the constructor is the initial capacity, then adds a new object to the array list, and finally checks that the addition was correctly performed by verifying that a subsequent index lookup operation returns the new object. (We omit visibility modifiers in all code fragments.)

```
[TestMethod]
void TestAdd() {
  ArrayList a = new ArrayList(0);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[0] == o);
}
```

It is important to note that unit tests include a test oracle that verifies the observed behavior with the expected result. By convention, the test oracle of a unit test is encoded using debug assertions. The test fails if any assertion fails or an exception is thrown. Unit test frameworks can also deal with expected exceptions, which in VSUnit are specified by additional custom attributes.

2.2 Parameterized Unit Tests

The unit test given above specifies the behavior of the array list by example. Strictly speaking, this unit test only says that adding a particular object to an empty array list results in a list whose first element is this particular object. What about other array lists and other objects?

```
[TestAxiom]
void TestAdd(ArrayList a, object o) {
  Assume.IsTrue(a!=null);
  int i = a.Count;
  a.Add(o);
  Assert.IsTrue(a[i] == o);
}
```

By adding parameters we can turn a closed unit test case into a universally quantified conditional axiom that must hold for all inputs under specified assumptions. Intuitively, the `TestAdd()` method asserts that for all array lists a and all objects o, the following holds:

$$\forall \texttt{ArrayList}\ a, \texttt{object}\ o.$$
$$(a \neq \texttt{null}) \rightarrow \text{ let } i = a.\texttt{Count} \text{ in } a.\texttt{Add}(o) \fatsemi a[i] == o$$

where '$\fatsemi$' denotes sequential composition from left to right, i.e. $(f \fatsemi g)(x) = g(f(x))$.

2.3 Test Cases

Which actual parameters must be provided to ensure sufficient and comprehensive testing? Which parameters can be chosen at all?

Consider again the `ArrayList` example. The `Add` method in the .NET Framework implementation distinguishes two cases. One occurs when the array list has enough room to add another element. (i.e. the array list's capacity is greater than the current number of elements in the array list). The other occurs when the internal capacity of the array list must be increased before adding the element.

If we assume that library methods invoked by the `ArrayList` implementation are themselves correctly implemented, we can deduce that running exactly two test cases is sufficient to guarantee that `TestAdd(...)` succeeds for all array lists and all objects. What are such two test cases?

```
[TestMethod]
void TestAddNoOverflow() {
  TestAdd(new ArrayList(1), new object());
}

[TestMethod]
void TestAddWithOverflow() {
  TestAdd(new ArrayList(0), new object());
}
```

Splitting axioms and test cases in this way is a *separation of concerns*. First, we describe expected behavior as PUTs. Then we study the case distinctions made by the code paths of the implementation to determine which inputs matter for testing.

2.4 Test Case Generation

We use symbolic execution to automatically and systematically produce the minimal set of actual parameters needed to execute a finite number of finite paths. Symbolic execution works as follows: For each formal parameter a symbolic variable is introduced. When a program variable is updated to a new value during program execution, then this new value is often expressed as an expression over symbolic variables. For each code path explored by symbolic execution. a *path condition* is built over symbolic variables. For example, the `Add`-method of the `ArrayList` implementation contains an *if*-statement whose condition is `this._items.Length == this._size` (where the field `_items` denotes the array holding the array list's elements and `_size` denotes the number of elements currently contained in the array list). The symbolic execution conjoins this condition to the path condition for the *then*-path and the negated condition to the path condition of the *else*-path. In this manner all constraints are collected, which are needed to deduce what inputs cause a code path to be taken.

Analysis of all paths can't always be achieved in practice. When loops and recursion are present, an unbounded number of code paths may exist. In this case we approximate by analyzing loops and recursion up to a specified number of unfoldings, similar to the heuristics used in [5]. Even if the number of paths is finite, solving the resulting constraint systems is sometimes computationally infeasible. Our ability to generate inputs based on path analysis depends upon the abilities of the constraint solver used; in our case we can use either Zap[19] or Simplify[8].

After collecting constraints for each code path, we use an automated solver to reduce the constraints collected in the previous step into concrete test cases.

When a constraint system cannot be solved automatically, the programmer must supply additional inputs. For example, when constructing suitable `ArrayList` values, which capacity should be picked and what elements should the array list contain?

There are two ways in which this information can be provided. The first is for the user to provide a set of candidate values for the formal parameters. In our running scenario let us assume that a user has provided the values

```
[TestValues(For="TestAdd", Parameter="a")]
ArrayList[] a = {new ArrayList(0),
                 new ArrayList(1)};

[TestValues(For="TestAdd", Parameter="o")]
object[] o    = {new object()};
```

Now the constraint solver can choose one of `a[0]`, `a[1]` and `o[0]` to obtain a solution that fulfills the constraints. The generated tests are:

```
TestAdd(a[0], o[0]);
TestAdd(a[1], o[0]);
```

The second way is to provide an invariant for a class that makes it possible to construct suitable instances using .NET reflection. The invariant is a Boolean predicate with the custom attribute `[Invariant]` attached to it. For array lists the invariant is

```
this._items != null && this._size>=0 &&
this._items.Length >= this._size
```

For the `TestAdd()` method, this invariant is instantiated with the symbolic variable `a` and serves as the initial path condition. This allows the constraint solver to give example input values for each symbolic variable encountered on each path. For the path with the condition `a._items.Length==a._size` the solver could choose the binding: `a._items.Length==0` and `a._size==0`. Using .NET reflection the tool can now produce an array list that corresponds exactly to `a[0]`.

In case no solution is found, the tool prints the path condition.

3. TOOL USAGE

We have developed the prototype tool Unit Meister which generates test cases from PUTs using symbolic execution and constraint solving [12].

Unit Meister's input is a .NET assembly containing PUTs in the form of test classes with axiom methods. If the number of code paths of the axiom methods and the called methods-under-test is infinite, bounds must also be given by the user, e.g. a time limit, or a bound on the number of loop and recursion unfoldings. Unit Meister then explores the code paths of a single specified axiom method or of all axiom methods. For each explored path, the output is either a test case if a solution of the path condition is found, or a representation of the path condition. Test cases can be represented as traditional closed unit tests in C#, or the generated concrete input data can be persisted in a table in a database, such that one row is created per explored code path and the entries of the row are the generated inputs.

Unit Meister can be invoked from the command line. We also provide an Add-In for Visual Studio 2005 Team System. In VSUnit, PUTs are written as "data driven" test methods. Such a test method is associated with a table in a database by means of a custom attribute. There are two modes of execution for such a test method:

- Symbolic execution with Unit Meister. All code paths within specified bounds are explored; when in any code path an unexpected exception is thrown or an assertion is violated, an error in the implementation (or the test method) has been found. For each code path, a test case is generated and persisted in the database if concrete inputs fulfilling the path condition can be obtained. The tree of symbolically explored paths can be visualized and inspected. Each leaf of the tree, which identifies a completed code path, is either decorated with concrete inputs representing a test case, or it is indicated that no concrete inputs were found for the path condition.

- Execution with VSUnit. After the test method has been explored by symbolic execution with Unit Meister as described above, the generated concrete test cases can be re-executed using VSUnit's test manager. Re-execution in this fashion uses the standard .NET runtime. VSUnit will invoke each test method once for each row of the database table, and the row's entries are used to instantiate the test method's parameters.

These two modes of execution can be seen as "record" and "replay" modes. When test cases are generated with Unit Meister, all possible code paths within the given bounds are exercised and checked for errors. At this point, re-executing the generated test cases with VSUnit will yield exactly the same results. When the implementation-under-test changes, the persisted test cases may no longer provide the same code coverage, since they were generated to cover a different implementation. New errors might not be discovered until Unit Meister explores the test method for the changed implementation again. However, generating test cases with symbolic execution is expensive; on the other hand, re-running previously persisted concrete test cases with VSUnit is an inexpensive way to quickly detect breaking changes.

4. RELATED WORK

The automatic generation of tests has recently received a lot of attention. Here we only try to cover those strands of research and tools that use symbolic evaluation for test case generation and which has influenced our work on parameterized unit testing.

Most work in the formal methods community concentrated on using models to generate black box tests for an implementation under test (IUT). Models can be property oriented, i.e. described by pre-/post conditions or functional programs, stateful i.e. described by some form of state machines, or intensional, i.e. described by axioms. In any case the goal of model-based test case generation is to derive test cases from the model until a certain model coverage is reached.

If the models are property oriented the models are typically analyzed symbolically to derive disjunctive normal forms. If the models contain recursion, then some kind of regularity or uniformity hypothesis is used that limits the number of unfoldings used to stop the test case generation process. A solution for the resulting formulas is then the test input for the IUT. This work goes back to [9], which proposed it for testing implementations described by VDM programs. It was recently reworked by [4], which uses Isabelle to formalize the test case generation process. In any case only one function at a time is tested.

Some recent frameworks also support symbolic generation of non-isomorphic complex object graphs, most notably TestEra[14] and Java PathFinder[21]. TestEra generates inputs from constraints given in Alloy, a first-order declarative language based on relations. The TestEra approach is still black-box testing, since the tests are not generated on the basis of the methods of the implementation. In the spirit of Korat[3], Java PathFinder constructs input object graphs lazily, observing a data structure invariant which must be written in Java in a special way to deal with partially initialized input. Only primitive field values can be symbolic, whereas object references are always concrete.

In parameterized unit testing we also use symbolic computation to derive test inputs, but we do not split conditions into their disjunctive normal forms, instead we simply use the path conditions as they are and we find covering inputs for the implementation methods and not the model. By working on the level of the implementation we find all (corner) cases. Further, if in our framework the user provides a data structure invariant as required for TestEra or Java PathFinder, then we can also use them to manufacture objects. If no or only a weak data invariant is given, we can still use it to bound the search space. In this case a set of representative objects must be given.

The purpose of test case generation from state based or property oriented formalisms is to generate sequences of method calls. Given a start and a final state, the BZ-TT tool uses constraint solving to derive start and final state covering method sequences from B specifications [1]. Closer in spirit to our work is the work done in testing from algebraic specifications. This work was started by Gaudel et al. [2]. They use axioms in various ways: to describe the test purpose; to obtain concrete data, which is necessary for the instantiations of the axioms; and to derive new theorems, which when tested should uncover errors in hidden state, i.e. state that is not represented by the model. For deriving those theorems they introduced regularity assumptions.

Another test goal is to test for robustness of individual methods. Usually this is done by random input data generation [6]. Recently [7] combined random input generation with constraint solving to test the robustness of individual methods, where ESC/Java [10] reports give constraint systems indicating necessary conditions for potential errors. These reports are often false positives, i.e. the constraints are unsatisfiable.

DART[11] also aims at testing methods of a program automatically for robustness, using a variation of symbolic execution where the program is explored exhaustively as long as the arising path conditions are in the realm of the used constraint solver; in all other cases, DART falls back on random input generation. The advantage is that no false-positives are generated, but in general no exhaustive testing within certain structural bounds can be achieved. In our framework, we test robustness all the time; we do however need parameter domain annotations to generate inputs and to avoid false-positives.

Acknowledgements

We thank Wolfgang Grieskamp for his contributions to the Exploring Runtime, XRT, and Thorsten Schütt for his work on a prototype symbolic execution framework. We thank Ward Cunningham, Brian Crawford, Mike Barnett, Colin Campbell, and Margus Veanes for many useful discussions. We are especially grateful to Tom Ball, Madan Musuvathi, and Shuvendu Lahiri, with whom we had many discussions that helped shape this work.

5. REFERENCES

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using contraint logic programming. In R. Hierons and T. Jerron, editors, *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 105–120. INRIA Report, August 2002.

[2] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[4] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2004.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[6] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[7] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *27th International Conference on Software Engineering*, May 2005. To appear.

[8] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, USA, 2003.

[9] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Industrial Strength Formal Methods, Formal Methods Europe (FME'93), Proceedings*, volume 670 of *LNCS*, pages 268–284. Springer, 1993.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.

[12] W. Grieskamp, N. Tillmann, and W. Schulte. XRT — Exploring Runtime for .NET — Architecture and Applications. In *Proc. 3rd SoftMC*, 2005. To appear.

[13] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[15] Microsoft. Visual Studio 2005 Team System. http://lab.msdn.microsoft.com/teamsystem/.

[16] J. W. Newkirk and A. A. Vorontsov. *Test-Driven Development in Microsoft .NET*. Microsoft Press, Apr. 2004.

[17] NUnit development team. NUnit. http://www.nunit.org/.

[18] Parasoft. Jtest manuals version 5.1. Online manual, July 2004. http://www.parasoft.com/.

[19] Testing, Verification and Measurement, Microsoft Research. Zap theorem prover. http://research.microsoft.com/tvm/.

[20] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. 13th International Symposium on Foundations of Software Engineering*, 2005. To appear.

[21] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.