Unit Tests Reloaded:

Parameterized Unit Testing with Symbolic Execution

Nikolai Tillmann, Wolfram Schulte
{nikolait, schulte}@microsoft.com

March 15, 2006

MSR-TR-2005-153

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

Unit tests are popular, but it is an art to write them in a way that specifies a program's behavior well and it is laborious to write enough of them to have confidence in the correctness of an implementation. Symbolic execution is an approach that can help. We describe techniques for unit testing based on symbolic execution. These techniques can be used to increase code coverage by finding relevant variations of existing unit tests, and they can be used to generate unit tests from an implementation automatically when no prior unit tests exist. The adoption of symbolic analysis techniques in commercial testing tools has already begun.

Introduction

Unit tests are becoming popular. A recent survey at Microsoft indicated that 79% of developers use unit tests [1]. Unit tests are written to document customer requirements, to reflect design decisions, to protect against changes but also, as part of the testing process, to achieve certain code coverage which in turns leads to a high confidence in the correctness of the program.

The growing adoption of unit testing is due to the popularity of methods like XP [2] ("extreme programming") and test execution frameworks like JUnit [3]. XP promotes Test-Driven Development (TDD) [4], where unit tests are written to guide the implementation of features. But these unit tests usually cover only very specific cases, and XP doesn't say when enough tests have been written. Test execution frameworks automate only test execution; they also do not automate the task of creating a comprehensive set of unit tests. Writing all unit tests by hand can be a laborious undertaking. In many projects in Microsoft there are more lines of code for the unit tests than for the implementation being tested. Are there ways to automate the generation of good unit tests?

We think the answer is yes, and this is the topic of the article. We describe *parameterized unit testing* with symbolic execution. Although first envisioned in 1976 [5], symbolic execution has only recently become feasible in practice. This advance is due to improvements in hardware and the development of better algorithms for automatic reasoning.

The techniques we describe can be used in several ways. We use symbolic execution to find inputs for parameterized unit tests that achieve high code coverage; we turn existing unit tests into parameterized unit tests; and we generate entirely new parameterized unit tests that describe the behavior of an existing implementation.

Parameterized unit tests describe behavior more concisely than traditional unit tests; in fact, parameterized unit tests can serve as specifications.

Both traditional testing and TDD benefit from these techniques because test inputs – including the behavior of entire classes – can often be generated automatically from compact parameterized unit tests.

In the rest of this article, we will introduce the concepts and illustrate the techniques with some examples. We assume that we test deterministic, single-threaded applications.

Symbolic execution has been implemented in many frameworks, e.g. as an extension to Java PathFinder [6] for Java, and XRT [7] for .NET.

What are unit tests?

A unit test is a self-contained program that checks an aspect of the implementation under test. Here is an example of a unit test that checks the interplay among .NET's `ArrayList` operations. The example is written in C#, omitting the class context. We omit visibility modifiers like `public` for brevity.

```
void AddTest() {
  ArrayList a = new ArrayList(1);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[0] == o);
}
```

First, the test creates an `ArrayList` with capacity 1. An array list is a container whose size may change dynamically. Internally, it uses a fixed-length array as backing storage. The array list's capacity is the allocated length of its current backing storage. Next, the test creates an object and adds it to the array list. Finally, the test checks that the array list at position 0 contains the newly inserted object.

Parameterized unit tests

Traditional unit tests do not take inputs. A natural extension would be to allow parameters. For example, the above test could be parameterized over the initial capacity of the array list:

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

This test is more general than the original test. Parameterized unit tests like this one can be called with various input values, perhaps drawn from an attached database.

Unit testing frameworks that support parameterized unit tests sometimes refer to them as data-driven tests (e.g. in [8]).

Parameterized unit tests separate concerns

Parameterized unit tests are a natural way to separate two concerns. On the one hand, they can be read as specifications. On the other hand, their instantiations give certain coverage of the code of the implementation.

Here is another parameterized version of the array list example that describes the normal behavior of the `Add` method with respect to two observers, the property `Count` and the indexing operator `[]`.

```
void AddSpec(ArrayList a, object o) {
  if (a != null) {
    int len = a.Count;
    a.Add(o);
    Assert.IsTrue(a[len] == o);
  }
}
```

Let’s look at the tested code. Figure 1 shows the implementation of .NET’s `ArrayList` class.

```
class ArrayList … {
  object[] _items;
  int _size, _version;

  ArrayList(int capacity) {
    if (capacity < 0) throw new ArgumentOutOfRangeException(…);
    _items = new object[capacity];
  }
  int Add(object value) {
    if (_size == _items.Length) EnsureCapacity(_size + 1);
    _items[_size] = value;
    _version++;
    return _size++;
  }
  void EnsureCapacity(int min) {
    if (_items.Length < min) {
      int newCapacity = _items.Length == 0 ? 16 : _items.Length * 2;
      if (newCapacity < min) newCapacity = min;
      …
      object[] newItems = new object[newCapacity];
      Array.Copy(_items, 0, newItems, 0, _size);
      _items = newItems;
      …
    }
  }
  object this[int index] {
    get {
      if (index < 0 || index >= _size) throw new ArgumentOutOfRangeException(…);
      return _items[index];
    }
    …
  }
  …
}
```
Figure 1: Relevant excerpt of .NET’s ArrayList implementation

The implementation of the `Add` operation has two code paths, one for the case where the capacity is sufficient to accommodate the new value and one for the case were additional storage must be allocated in `EnsureCapacity`.

If we call the `AddSpec` method shown above with inputs

```
AddSpec(new ArrayList(0), new object());
AddSpec(new ArrayList(1), new object());
```

we execute exactly those two code paths. Assuming that the contract for `EnsureCapacity` is obeyed, i.e., `EnsureCapacity` guarantees that the `_items` array is resized so its length is greater or equal `_size + 1` (and not considering possible integer overflows or memory allocation exceptions), the assertion embedded in `AddSpec` holds for all array lists and all objects. Note that we don't need any other input to test `Add`, since any other input will execute exactly the same paths as the two inputs mentioned above.

Automatically generating unit tests

Techniques for automating test case generation are a research topic with practical relevance. In this article we present techniques that we implemented in the prototype tools UnitMeister [9] and AxiomMeister [10]. The tools can

- Generate unit tests. Given an implementation it is possible to generate parameterized unit tests that describe its behavior. Such tests can be used when refactoring code and they help in program understanding.
- Generalize existing unit tests. If we already have unit tests, we can generalize them to parameterized unit tests.
- Instantiate parameterized unit tests. If we have parameterized unit tests, there are often ways to automatically deduce sensible inputs that guarantee maximal coverage with a minimal number of test cases.

Figure 2 gives an overview of how these topics are related.

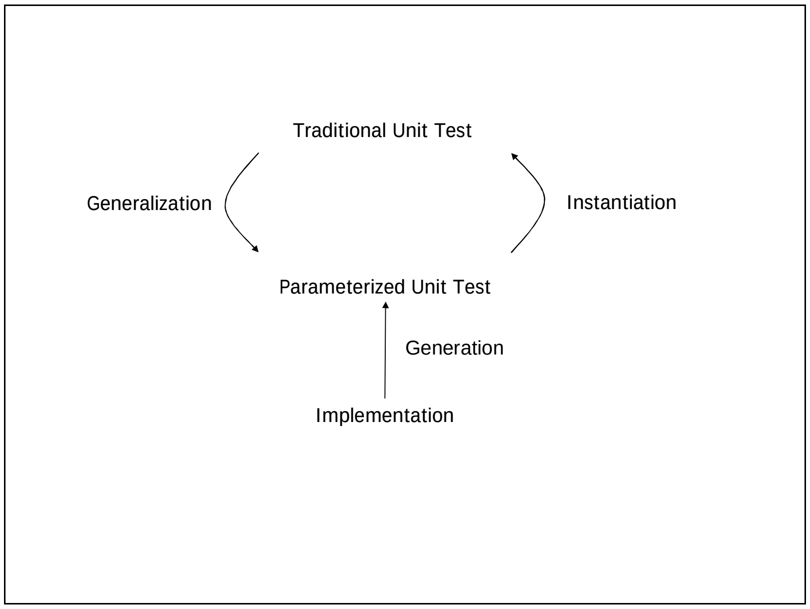

Figure 2: Connections between traditional and parameterized unit tests.

Symbolic execution

S*ymbolic execution* is a way to analyze the behavior of a program for all possible inputs. Instead of supplying the normal inputs to a program (e.g. concrete numeric values) one supplies symbols that represent arbitrary values. Symbolic execution proceeds like normal execution except that the values computed may be expressions over the input symbols.

Symbolic execution builds a *path condition* over the input symbols. A path condition is a mathematical formula that encodes data constraints that result from executing a given code path. For example, recall the `Add` method of the .NET `ArrayList` class given above. When symbolic execution reaches the *if*-statement, it will explore two execution paths. The *if*-condition is conjoined to the path condition for the *then*-path and the negated condition to the path condition of the *else*-path. Figure 3 shows the path conditions of the **`ParameterizedAddTest`** method we saw earlier.

Finding concrete input values that satisfy each path condition provides full *path coverage* with the minimum number of test inputs. If the number of paths is finite and the test passes for test inputs for each path, we have in fact *proven* the implementation for the parameterized unit test: It cannot fail for any inputs.

Symbolic execution systematically unfolds loops and recursion. The resulting number or length of paths might be too large to analyze (or even infinite). In general the existence of unbounded numbers of execution paths due to loops and recursion is a limitation of symbolic execution. However, approximation techniques can be applied. For example, it is possible to analyze loops and recursion up to a fixed number of iterations.

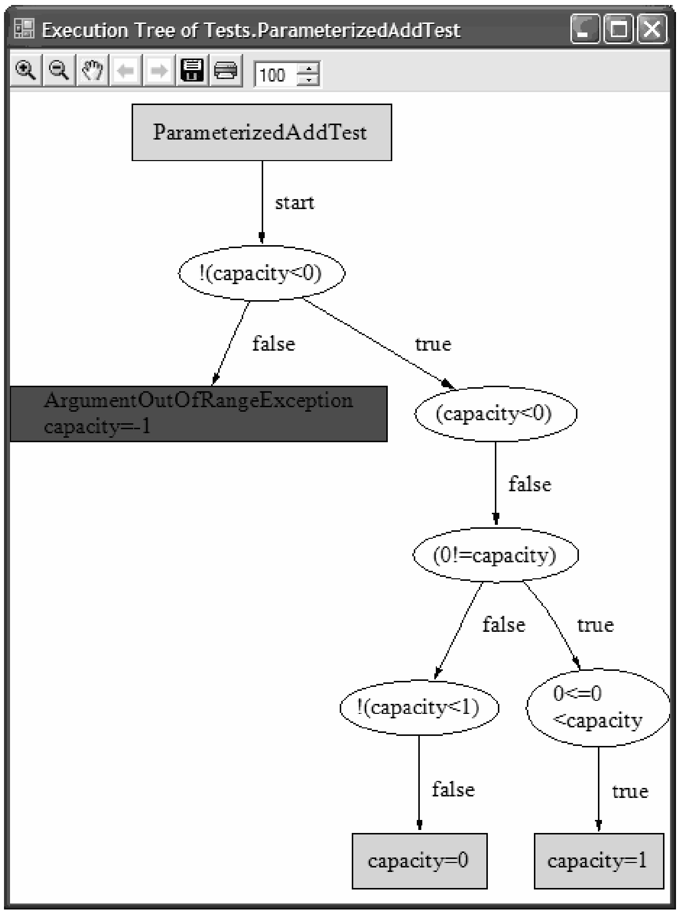

Figure 3. UnitMeister's visualization of possible execution paths for for **`ParameterizedAddTest`**. In the tree, the condition `(0 != capacity)` is the condition of the above *if*-condition in terms of the test's symbolic input `capacity`. The ovals show the conditional branch points encountered on each execution path. The outgoing edges represent possible evaluations of the conditions. The rectangles represent path terminations with exemplary concrete assignments.

Deciding feasibility

A path is *infeasible* if no concrete inputs exist that would cause this path to be taken. In such cases the path condition is self-contradicting. Such infeasible paths can be pruned away during symbolic execution.

Constraint solving and automatic theorem proving techniques can often decide whether a path condition is feasible. For example, efficient decision procedures for linear arithmetic problems exist. If the implemented decision procedures cannot decide the feasibility of a path condition , we can either prune the path, which leads to an under-approximation of the possible behaviors of the program (not all possible behaviors will be found) or we can include it in the exploration, which leads to an over-approximation (execution paths are considered which cannot arise in reality).

Reusing parameterized unit tests in symbolic execution

Decision procedures often simplify expressions according to certain *rules* in order to better reason about them. For example, *a* + 0 simplies to *a*.

Parameterized unit tests can be interpreted as rules, too! We can rewrite our unit test mathematically as a universally quantified conditional expression. A universally quantified expression says that for all *x* some Boolean condition p(*x*) holds. For example, the `AddSpec` says that for every `ArrayList` *a* and `object` *o*, we have

(*a* == null or let *len* be *a*.Count in (*a*.Add(*o*) followed by *a*[*len*] == *o*)).

Once the validity of a parameterized unit test has been established we can add it as a rule. As a consequence, symbolic execution does not have to execute the code of the `Add` method and the index operator anymore, but it can treat the array list operations like integer operations when building up and reasoning about constraints. When software is designed as a layered system, this technique can help make symbolic execution scale.

Instantiation of parameterized unit tests

The goal of instantiating parameterized unit tests is to obtain concrete test cases with high coverage of the tested implementation. How many instantiations should we consider? And which values should be chosen to instantiate a parameterized unit test?

Symbolic execution can help in answering those questions: it yields a set of paths characterized by their path conditions. We need only consider a set of paths that gives us the coverage we want. If we want path coverage we have to execute all feasible paths of the implementation. If we want to have a weaker coverage like branch coverage we limit symbolic execution so that it only tries to cover branches. In any case, for each chosen path we need only one *instantiation* fulfilling the path condition, since we know that all instantiations follow exactly the same code path.

We use constraint solving techniques to find instantiations of path conditions. Constraint solvers often operate over finite domains, like integer ranges. This means that in the case of parameterized unit tests the user must provide a set of values to choose from. As an alternative to exact solutions, random values can be chosen as an approximation. For example, Cute [11] uses random inputs when constraint solving fails.

Constraint solvers are not well suited to create objects and bring them into a desired state, so we have developed a technique to deal with objects, too.

Symbolic mock objects

In testing, *mock objects* are often used to imitate the behavior of actual software components. A mock object has a lightweight implementation that realizes only a fraction of the full functionality of the software component it impersonates. But what should be the behavior of a mock object and how many behaviors do we have to consider? Can we generate mock objects automatically?

Symbolic mock objects provide an answer. For a well defined interface a symbolic mock type can be synthesized to represent an arbitrary implementation of the interface. The behavior of a symbolic mock object is not fixed in advance. In a manner similar to introducing symbols for the parameters of a unit test, we introduce a new symbol to represent the result of each call to a method of a mock object; we can compute concrete mock objects by solving the path conditions involving this symbol.

Unconstrained mock objects

Unconstrained mock objects often allow the symbolically executed code to behave in an unexpected way. They obey the type system, but they may not respect other, often implicit, behavioral assumptions.

Consider the following example:

```
interface INumberProvider {
  int GetNumber();
}

int Compute(INumberProvider x) {
  return Math.Abs(10 / (x.GetNumber() + x.GetNumber()));
}
```

Let's suppose that `Compute` should return a non-negative result. We can express this constraint as a parameterized unit test:

```
void ComputeSpec(INumberProvider x) {
  int result = Compute(x);
  Assert.IsTrue(result >= 0);
}
```

We can symbolically execute **`ComputeSpec`** with a symbolic mock object for the parameter **`x`**. To represent the result of each method call on a symbolic mock object, symbolic execution introduces a new symbol. Since **`ComputeSpec`** calls **`Compute`**, which calls **`GetNumber`** twice on **`x`**, symbolic execution creates two symbols. Again ignoring integer overflows, the parameterized unit test gives rise to two execution paths: for the normal execution the sum of the results of the two **`GetNumber`** calls must be greater or equal 0; for the exceptional execution the result of the two calls must be 0, which triggers a division by zero exception error.

Here are the generated implementations of the mock objects: one for the normal execution, one for the exceptional execution. The internal **`_GetNumber_counter`** distinguishes the different calls to **`GetNumber`** from each other.

```
class MockClass0 : INumberProvider {
  static int _GetNumber_counter;
  int IComponent.GetNumber() {
    switch (_GetNumber_counter++) {
      case 0: return 713;
      case 1: return -9;
      default: throw new InvalidOperationException();
    }
  }
}

class MockClass1 : INumberProvider {
  static int _GetNumber_counter;
  int IComponent.GetNumber() {
    switch (_GetNumber_counter++) {
      case 0: return 0;
      case 1: return 0;
      default: throw new InvalidOperationException();
    }
  }
}
```

Note that **`MockClass1`** will cause the unit test to fail. But in unit testing, mock objects are often introduced to substitute only "friendly" components. We can adapt symbolic execution correspondingly: when during symbolic execution a failure can be traced back to a return value of a supposedly friendly mock object, such a failure is simply pruned. For example, the **`MockClass1`** wouldn't be generated if we only want to have friendly mock objects.

Constrained mock objects

It is often desirable to restrict the degrees of freedom available to a mock object. For example, a hash table only functions as expected if the keys follow certain rules regarding the properties and interplay of the `GetHashCode` and `Equals` methods. We can write this requirement as a parameterized unit test:

```
void GetHashCodeEqualsSpec(object x, object y) {
  if (x != null && x.Equals(y))
    Assert.IsTrue(x.GetHashCode() == y.GetHashCode());
}
```

As before, this parameterized unit test can be interpreted as a mathematical formula and added to the rules of a theorem prover. The theorem prover can assist in making sure that mock objects obey the rules. We call such restricted mock objects *constrained*.

Generalization of existing unit tests

Many unit tests exist already. Generalization is a way to automatically turn these into parameterized unit tests which are more expressive and achieve higher code coverage. The idea behind generalization is to refactor the unit test by promoting the concrete values that appear in the body of the test to parameters. As discussed above, symbolic execution finds possible execution paths through the generalized test.

Different execution paths might be possible. Some may lead to failures. We distinguish two kinds of failures:

- *Failures which can be corrected by adjusting the test text*. For example, when an assertion fails, it can simply be removed, and when an unexpected application exception is thrown, a try-catch block can be introduced. This is a simple form of specification inference.
- *Failures which indicate a programming error like a division-by-zero error*. These errors result from violations of implicit preconditions of the implementation. We treat such cases like successful execution paths, for which we do not adjust the test text. Executing these paths' instantiations will exhibit the errors.

Let us look at the `AddTest` method given above. We promote the values `1` and `0` to test parameters `x` and `y` respectively. Using symbolic execution, our tool finds all cases which completely cover the feasible execution paths of the tested implementation that do not result in an exception and pass the assertion. As we already know, there are two cases depending on the choice of the initial capacity. The tool chooses solutions for `x` and `y`.

Instantiations for GeneralizedAddTest0	X	y
Case 1 (enough capacity from beginning)	1	0
Case 2 (insufficient initial capacity)	0	0

```
void GeneralizedAddTest0(int x, int y) {
  ArrayList a = new ArrayList(x);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[y] == o);
}
```

Three cases cover all failures. In each failure case, the `ArrayList` implementation throws an `ArgumentOutOfRangeException`, either in the constructor or in the indexing operator `[]`. Our tool changes the test text by adding a try-catch block.

Instantiations for GeneralizedAddTest1	x	y
Case 1 (capacity must be non-negative)	-1	0
Case 2 (index below lower bound)	1	-1
Case 3 (index above upper bound)	1	1

```
void GeneralizedAddTest1(int x, int y) {
  try {
    ArrayList a = new ArrayList(x);
    object o = new object();
    a.Add(o);
    Assert.IsTrue(a[y] == o);
  } catch (ArgumentOutOfRangeException) { }
}
```

Generalization is not limited to integer values. Newly created objects, e.g. `new object()`, can be promoted to parameters, too. They will then be instantiated with mock objects.

Automatic generation of parameterized unit tests

Generalization, as described in the previous section, is limited: It does not change the structure of the unit test, since it only substitutes values in an otherwise fixed program text. Writing tests that exercise the relationships between methods remains an art.

Again, we can use symbolic execution to find relationships between methods. The core idea is to explore possible execution paths not only with *symbols for parameters* but also starting from a *symbolic state*.

In a symbolic state each field is initialized with a special symbol that is constrained only by the field’s type and known data invariants. We can use symbolic analysis to determine possible execution paths of each *modifier* method (i.e., each method that updates program state as opposed to just reading it) of a given program. Each execution path terminates in a state where the fields’ final values are represented by expressions over the input symbols.

Thus, for each path, we get a description of how the final values are derived from the input symbols. This description can be seen as path-specific mappings of inputs to outputs (where the initial symbols of the fields are the inputs and the final expressions are the outputs).

The following table describes such mappings as they result from symbolic execution of the `ArrayList`'s `Add` method (we omit rare cases such as potential out-of-memory exceptions to simplify the presentation). We use the expression *e*′ to denote the value of the expression *e* at the end of the `Add` method, the symbol `o` to represent the method's parameter, and `_capacity` as a shorthand notation for `_items.Length`.

Mappings of inputs to outputs for `Add` method execution paths		
Path	Path Condition (applies when)	Input/Output Mapping
(common to all paths)	`this != null && _items != null && 0 <= _size < _capacity′`	`_size′ == _size+1 && _version′ == _version+1 && _items[_size]′ == o`
sufficient capacity	`_size < _capacity`	`_capacity′ == _capacity && _items′ == _items`
no capacity	`_size == 0 && _capacity == 0`	`_capacity′ == 16 && _items′ == new object[16]`
For every positive integer *i*, insufficient capacity of *i* elements	`_size ==` *i* `&& _capacity ==` *i*	`_capacity′ == _capacity*2 && _items′ == new object[`*i*`] && _items[0]′ == _items[0] && ... && _items[`*i*`-1]′ == _items[`*i*`-1]`

The path condition and input/output mapping reflect internal implementation details. In this exaple, the number of paths is unbounded. We have one path for every positive integer; these paths arise from unfolding a loop in `Array.Copy`. We need summaries of the path conditions and input/output mappings that abstract away implementation details.

Observer methods (i.e., methods of the program that read program state but do not update it) provide the summaries we need. Observers group initial and final states into equivalence classes that behave in the same way as far as the observers are concerned. In other words, they group exactly all those execution paths together which differ only in unobservable implementation details.

Which and how many observers will make good summaries? We can answer that question if you accept the following statement: the implementation should be *observably deterministic*; that is, if we can observe a difference in summarized final states, a difference should have existed in the summarized initial states.

Our tool checks this property for every pair of paths. If it is violated the user is notified; this usually indicates that the design is either incomplete (observers cannot detect differences in initial states) or too verbose (observers allow detection of implementation details in final states).

Consider the `Add` method of the `ArrayList` class. If we only consider the property `Count` that returns the value of the `_size` field as the observer, we can note that all execution paths increment `_size` by one. Since all paths of `Add` are indistinguishable under the `Count` property, our tool summarizes them with a single parameterized unit test:

```
void AddCountSpec(ArrayList a, object o) {
  if (a != null) {
    int oldCount = a.Count;
    a.Add(o);
    Assert.IsTrue(a.Count == oldCount + 1);
  }
}
```

In addition to observations directly provided by observer methods, new observations arise from combinations of observers. When we consider `Count`, `Capacity` (which returns `_items.Length`) and the relation `>` ("greater-than") as observers, our tool finds that all paths can be divided into two groups and summarizes them follows:

```
void AddCountCapacitySpec1(ArrayList a, object o) {
    if (a!=null &&
        a.Count == a.Capacity) {
        int oldCount = a.Count;
        int oldCapacity = a.Capacity;
        a.Add(o);
        Assert.IsTrue(a.Count == oldCount + 1 &&
                      a.Capacity > oldCapacity);
    }
}

void AddCountCapacitySpec2(ArrayList a, object o) {
    if (a!=null &&
        a.Count != a.Capacity) {
        int oldCount = a.Count;
        int oldCapacity = a.Capacity;
        a.Add(o);
        Assert.IsTrue(a.Count == oldCount + 1 &&
                      a.Capacity == oldCapacity);
    }
}
```

Generated parameterized unit tests reflect the behavior of the implementation. Whether they exhibit the *intended* behavior must be checked by the user. Even so, such inferred summaries have many uses: they help in program understanding and code reviews; they can serve as regression tests; and they can be used as specifications in program verification systems like Spec# [12].

Experimental Results

UnitMeister [9] implements the instantiation technique and has recently been extended to mock object generation and unit test generalization. AxiomMeister [10] generates parameterized unit tests.

We evaluated UnitMeister on a small set of common benchmarks. We wrote parameterized unit tests for some collection types of the .NET 1.1 base class library and other collection types from an early version of the Spec# programming system. The tests are straightforward encodings of the available documentation. UnitMeister symbolically explored the tests until full coverage of the reachable branches of the tested methods was achieved. The following table shows how many methods were tested, how long it took UnitMeister to explore a certain number of paths, and how many bugs it found. We considered any discrepancy between observed and documented behavior to be a bug.

Class	Methods Tested	Unit Tests	Paths Explored	Bugs found	Time/s
`ArrayList`	10	12	34	1	4
`Enumerator`	4	10	67	1	10
`Hashtable`	9	11	30	0	30
`LinkedList`	3	3	64	1	4
`RedBlackTree`	3	3	457	0	427

The following table shows the results of applying AxiomMeister to some collection types of the .NET base class library. For every class, the table shows the sum of the considered modifier and observer methods and the time of symbolically exploring a number of paths and reducing them to a number of parameterized unit tests. Most of the generated tests resemble the specifications we would write by hand.

Class	Methods Considered	Paths Explored	Unit Tests Generated	Time/s
`Stack`	6	7	6	2
`ArrayList`	13	142	26	29
`Hashtable`	9	835	14	276

For both UnitMeister and AxiomMeister the results show that the techniques presented here can be successfully applied on nontrivial implementations like a hash table. The automatic analysis of hand-written parameterized unit tests exposed several bugs.

Conclusion

Symbolic analysis techniques have been applied successfully on large, real-world programs. For example, PREfix [13] is applied on the Microsoft Windows code base. Related techniques have been recently adopted by companies like Parasoft [14] and Agitar [15], which produce commercial tools for unit testing.

We are currently applying the presented techniques on products being developed at Microsoft. We assume that reusing parameterized unit tests and especially symbolic mock objects will play a central role for the scalability of parameterized unit testing. Our first experiences confirm this [9], but further experiments are necessary.

Acknowledgements

Our parameterized unit testing tools would not exist without the symbolic execution framework XRT, which was co-developed with Wolfgang Grieskamp. We thank Tom Ball for many insightful discussions, and Colin Campbell for his critical review of this article.

Bibliography

[1] Venolia, G., DeLine, R., and LaToza, T. Software Development at Microsoft Observed. Microsoft Research Technical Report MSR-TR-2005-140. October 2005.
[2] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2001.
[3] Gamma, E. and Beck, K. JUnit: A Regression Testing Framework. http://www.junit.org, 2001.
[4] Beck, K. Test-Driven Development: By Example. Addison-Wesley, 2002.
[5] J. C. King. Symbolic execution and program testing. Commun. ACM, 19(7):385–394, 1976.
[6] 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.
[7] Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT - Exploring Runtime for .NET - Architecture and Applications. In Proc. SoftMC 2005: Workshop on Software Model Checking, July 2005.
[8] Microsoft. Visual Studio 2005 Team System. http://lab.msdn.microsoft.com/teamsystem/
[9] Tillmann, N. and Schulte, W. 2005. Parameterized unit tests. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering (September 05 - 09, 2005). ESEC/FSE-13. ACM Press, New York, NY, 253-262.
[10] Cheng, F., Tillmann, N. and Schulte W. Discovering Specifications. Microsoft Research Technical Report MSR-TR-2005-146. October 2005.
[11] Sen, K., Marinov, D., and Agha, G. 2005. CUTE: a concolic unit testing engine for C. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering (September 05 - 09, 2005). ESEC/FSE-13. ACM Press, New York, NY, 263-272.
[12] Barnett, M., Leino, R., Schulte, W. The Spec# Programming System: An Overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.
[13] Bush, W.R., Pincus, J.D., Sielaff, D.J. A Static Analyzer for Finding Dynamic Programming Errors. Software Practice and Experience. Volume 30, June 2000.
[14] Parasoft Corporation. http://www.parasoft.com/
[15] Agitar Software. http://www.agitar.com