Parameterized Unit Tests

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. Parameterized unit tests separate two concerns: 1) They specify the external behavior of the involved methods for all test arguments. 2) Test cases can be re-obtained as traditional closed unit tests by instantiating the parameterized unit tests. Symbolic execution and constraint solving can be used to automatically choose a minimal set of inputs that exercise a parameterized unit test with respect to possible code paths of the implementation. In addition, parameterized unit tests can be used as symbolic summaries which allows symbolic execution to scale for arbitrary abstraction levels. We have developed a prototype tool which computes test cases from parameterized unit tests. We report on its first use testing parts of the .NET base class library.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications — *Methodologies*

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 parameters. It represents a 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, and the Extreme Programming discipline [20] for instance leverages them to permit easy code changes. 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 [24] (VSUnit), to automatic unit-test generation, e.g. as provided by Parasoft's JUnit Test Tool [27]. However these tools don't provide any guidance for:

- which tests should be written (for internal and for external behavior),
- 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 (including the methods-under-test it calls) 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, there is a mathematical proof that the PUT would pass for all possible instantiations; a result which goes back to [21]. 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.

In summary, PUTs' *contributions* are:

- They allow unit tests to play a greater role as specifications of program behavior. In fact, PUTs are axiomatic specifications.
- They enable automatic case analysis, which avoids writing implementation-specific unit tests.
- Their generated test cases often result in complete path coverage of the implementation, which amounts to a formal proof of the PUTs' assertions.

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

We have developed the prototype tool Unit Meister, which uses symbolic execution of .NET assemblies in the Exploring Runtime, XRT[17], to instrument PUTs:

- It allows 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.
- Evaluations have shown Unit Meister's ease-of-use and usefulness.

We are working on a plan to integrate the methodology of PUTs into the forthcoming Visual Studio Team System product. So far we have mainly applied PUTs on single-threaded abstract data types such as those provided by the .NET Framework base class library.

Section 2 illustrates the advantages of PUTs on some examples. Section 3 discusses the formal framework and sketches our implementation. Section 4 evaluates the results achieved so far. Section 5 presents related work and Section 6 provides concluding remarks.

2. OVERVIEW

2.1 Traditional Unit Tests

Using the conventions of NUnit[25, 26] and VSUnit[24], 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 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 compares observed behavior with expected results. By convention, the test oracle of a unit test is encoded using 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, see also Section 3.4.

2.2 Parameterized Unit Tests

The unit test above specifies the behavior of the array list by example. Strictly speaking, this unit test only says that by adding a new object to an empty array list, this object becomes the first element of the list. 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 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:

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

where '**;**' denotes sequential composition from left to right, i.e. $(f\ \texttt{;}\ g)(x) = g(f(x))$. Section 3.4 explains how axiom formulas are derived from PUTs in more details. See also [5] for an overview of the theory and practice of algebraic specifications.

2.3 Test Cases

Adding parameters to a unit test improves its expressiveness as a specification of intended behavior, but we lose concrete test cases. We can no longer execute this test axiom by itself. We need actual parameters. But which values must be provided to ensure sufficient and comprehensive testing? Which values can be chosen at all?

In the `ArrayList` example, if we study the internal structure of the .NET Framework implementation, we observe that there are two cases of interest. One occurs when adding an element to an array list that already has enough room for the new 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.

```
[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 make sense 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 [9]. 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[28] or Simplify[12].

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 simply check if `a[0]`, `a[1]` or `o[0]` satisfy 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 system can now produce an array list that corresponds exactly to `a[0]`.

If no solution is found, the tool prints the path condition.

2.5 Reusing Parameterized Unit Tests

While symbolic execution is more general than concrete execution, it is also slower. The number of possible paths to consider can grow exponentially with the number of control flow decisions. We need a way to prune the search space.

Consider the following example of a bag class. A bag is an unordered collection of values, i.e. elements may appear more than once in a bag. We implement a bag by using a hash table in which an element and its multiplicity are stored as key-value pairs.

```
class Bag {
  Hashtable h = new Hashtable();

  void Add(object o) {
    if (h.ContainsKey(o)) h[o] = (int)h[o] + 1;
    else h[o] = 1;
  }

  int Multiplicity(object o) {
    if (h.ContainsKey(o)) return (int)h[o];
    else return 0;
  }
}
```

A test for our bag might be that `Add(x)` increments `x`'s multiplicity. We specify this as follows:

```
[TestAxiom]
void AddMultiplicityTest(Bag b, object x) {
  Assume.IsTrue(b!=null & x!=null);
  int m = b.Multiplicity(x);
  b.Add(x);
  Assert.IsTrue(m+1 == b.Multiplicity(x));
}
```

When this unit test is symbolically executed, two cases arise in the bag implementation: One where the added object is already contained in the hash table, and one where it is not. However, executing the hash table implementation induces many more case distinctions (whether the capacity should be increased, whether a hash collision must be handled, etc.), and as a result we get dozens of distinct code paths. On the other hand, if we also have axioms summarizing the hash table's operations, we can avoid execution of the hash table by introducing expressions representing calls to the hash table and using the hash table's axioms as rewrite rules on those expressions. As a result, the hash table's implementation specific cases no longer need to be explored. Less time is needed for the analysis, and only two test cases specific to the actual bag implementation will be generated.

Again, we use .NET custom attributes to indicate that certain axioms should be reused. In the following example axioms defined in the `HashtableTests` class should be reused during symbolic execution of `BagTest`.

```
[TestClass, UsingAxioms(typeof(HashtableTests))]
class BagTests {
  [TestAxiom]
  void AddMultiplicityTest( ... ) { ... }
  ...
}
```

For this example, we need two hash table axioms that relate the constructor, `ContainsKey` and the indexer. We use the custom attribute `ProvidingAxioms` to indicate which type is summarized.

```
[TestClass, ProvidingAxioms(typeof(Hashtable))]
class HashtableTests {
  [TestAxiom]
  void NothingContainedInitially(object key) {
    Assume.IsTrue(key!=null);
    Hashtable h = new Hashtable();
    Assert.IsTrue(!h.ContainsKey(key));
  }

  [TestAxiom]
  void SetImpliesContainsAndGet(
         Hashtable h, object key, object val) {
    Assume.IsTrue(h!=null && key!=null);
    h[key] = val;
    Assert.IsTrue(h.ContainsKey(key));
    Assert.IsTrue(h[key] == val);
  }
  ...
}
```

Later we will see that we actually need two more axioms which specify that the externally observable state of the hash table is not changed by any of its observers. We defer the introduction of these axioms to Section 3.4.

Using universally quantified axioms in the context of symbolic exploration solves another problem, too. Often parts of a program might not yet be implemented, and sometimes the implementation cannot be interpreted by the symbolic execution engine. Since we

cannot explore code that does not exist or that cannot be interpreted, we need a different description of its behavior. And the best way to describe it is via PUTs, i.e. as axioms! In fact, we already used implicit axioms during the generation of test cases for the `ArrayList` class, which uses arrays, and the arrays of .NET are implemented outside of the .NET Framework.

3. FRAMEWORK

We next formalize the notions introduced informally in the previous section. First we describe our representation of symbolic states, constraints and their evolution. Next we describe how to derive and consume axioms from PUTs.

We also sketch our current implementation. It rests on two components: a backtrackable interpreter for .NET's intermediate instruction language (CIL) and a theorem prover, either Simplify[12] or Zap[28]. The interpreter operates on register-based instructions which are directly derived from CIL. The interpreter is optimized to deal efficiently with concrete and symbolic data representations. The theorem provers are used for reasoning about the feasibility of constraints and aid in finding concrete solutions.

3.1 Symbolic State

Symbolic states are like concrete states on which a conventional program execution operates, except that symbolic states can contain expressions with symbolic variables.

Symbolic expressions

Let *ObjectId* be an infinite set of potential object identifiers, *VarId* a set of symbolic variable identifiers, *TypeId* a set of type identifiers, and *FuncId* a set of function symbols, such that these sets are mutually disjoint. The set of untyped *symbolic expressions E* is described by the following grammar where the meta-variables o, v, t and f range over *ObjectId*, *VarId*, *TypeId* and *FuncId*, respectively.

E	$=$	o	object ids
	$\mid$	v	variables
	$\mid$	t	types
	$\mid$	$f(\bar{E})$	function application
	$\mid$	$\forall \bar{v}.E$	universal quantification

We use vector notation $\bar{x}$ to denote lists of items $x_1, \ldots, x_n$.

An expression is *ground* if it does not contain symbolic variables.

Function symbols

We distinguish two classes of function symbols:

- *Predefined function symbols* have a fixed interpretation in the theorem provers used. For instance, *add* denotes integer addition, and *and* denotes logical conjunction; *equals*(x, y) denotes whether x and y represent the same value for value types, or the same object identity for reference types, and *subtype*(x, y) establishes that x is a subtype of y. The literals `null`, `void`, 0, 1, 2 are nullary functions.

  Also supported are storage function symbols operating on maps. A map is an extensionally defined finite partial function. *empty* denotes the empty map; *update*(m, x, y) denotes the update of map m at index x to the new value y; *select*(m, x) selects the value of map m at index x.

  Some of these functions are partial. Their application will always be guarded by constraints imposed on the state, as will be described in Section 3.2.

  In the following, we often write $x = y$ for *equals*(x, y), and $x \wedge y$ for *and*(x, y). We omit $()$ for function applications on empty tuples.

- *Uninterpreted function symbols* represent properties of objects and method calls appearing in axioms.

  For example, $type(x)$ denotes the runtime type of object x, and *len*(x) the length of array x. *field*$_f(x)$ denotes the address of field f of object x, *elem*(x, y) the address of the array element at index y of array x. Expressions denoting addresses of object fields and array elements are used as indices in applications of storage functions.

  For each method m of the program with n parameters (including the `this` parameter for instance methods) we introduce up to three uninterpreted function symbols which are used to summarize different dynamic aspects of m: m_s, m_x, and m_r. Each of these functions takes $n + 1$ parameters, where the additional first parameter represents the state of the heap just before an invocation to m. Let h be an expression denoting the state of the heap in which $m(\bar{x})$ is called. Then $m_s(h, \bar{x})$ denotes the resulting state of the heap, $m_r(h, \bar{x})$ denotes the return value of the call, if any, and $m_x(h, \bar{x})$ represents the type of an exception that m throws, or `void` if no exception can be thrown.

Heaps

We distinguish two kinds of heaps.

- The *extensional heap* is represented by nested applications of the *update* function, indexed by field and array element addresses only. *empty* denotes the initial extensional heap. For example, the execution of the code fragment

  ```
  p.f=1; q.g=2;
  ```

  turns a given extensional heap H_e into the following extensional heap H'_e, assuming the locations `p`, `q` hold the expressions t, u respectively, and t and u cannot evaluate to `null`.

  $$H'_e = \mathit{update}(\mathit{update}(H_e, \mathit{field}_{\mathtt{f}}(t), 1), \mathit{field}_{\mathtt{g}}(u), 2)$$

  If an extensional heap expression contains symbolic variables, then the expression actually describes many concrete heaps, possibly isomorphic.

- The *intensional heap* is described by a history of method invocations: *initial* denotes the intensional heap where no method has been called yet. $m_s(H_i, \bar{x})$ represents the sequence of method calls encoded in the intensional heap expression H_i, followed by a call to $m(\bar{x})$. Consider for example the execution of the following code fragment.

  ```
  ArrayList a = new ArrayList(); a.Add(o);
  ```

  This code fragment turns a given intensional heap H_i into H'_i, where a is a fresh object identifier, and t the expression held in location `o`.

  $$H'_i = \mathtt{Add}_s(\mathtt{ArrayList}_s(H_i, a), a, t)$$

  Note that constructors do not create new objects, but are seen as methods that are called on fresh object identifiers.

Usually, we partition the types and their methods of the program to work on either heap. We discuss finer partitionings in Section 5.

Symbolic state

A *symbolic state* is a 5-tuple $S = (O, A, H_e, H_i, X)$, where the current set of objects O is a subset of *ObjectId*, the program stack A is a stack of activation records, H_e and H_i are expressions denoting the extensional heap and the intensional heap respectively, and

finally, X, an object expression, denotes the current exception. We associate with each activation record a method, a program counter pointing to the current instruction to execute in the method, as well as a store for the formal parameters and local variables. See [6] for a typical formalization of activation records. We say a computation in state S is normal if X is `null`, a computation is abrupted if X denotes an exception object. We write $O(S)$, $H_e(S)$, etc. for projections on S. We write S^{+1} for the state which is like S except that the program counter has been incremented. The set of all symbolic states is called *State*.

3.2 Constraints

A *constraint* on a symbolic state is a pair $C = (BG, PC)$, where BG is the static background, which only depends on the program declarations, and PC is the dynamic path condition, which is built up during symbolic evaluation.

The background conjoins $subtype$ predicates, encoding the type hierarchy, and axioms, whose generation is described in Section 3.4. Assume the whole program consists only of the class definition `class C {}`, then the background would just consist of the single predicate *subtype*(`C`, `System.Object`).

We write $BG(C)$ and $PC(C)$ for projections on C, and $(BG,\ PC) \wedge c$ for $(BG,\ PC \wedge c)$. The set of all constraints is called *Constraints*.

A constrained state is a pair (S, C).

Let D be a non-empty set, I_0 an interpretation of (S, C) that maps every n-ary function symbol appearing in S or C to an n-ary function over $D^n \to D$, and σ denote an assignment of the symbolic variables appearing in (S, C) to elements in D. With I we denote the usual meaning function which maps symbolic expressions with symbolic variables to values in D [22].

We call an interpretation I of symbolic expressions appearing in (S, C) to ground expressions a *solution* of (S, C) if I is a model for $BG \implies PC$. If a solution for (S, C) exists, we say that (S, C) is *feasible*.

3.3 Symbolic Evaluation

We next discuss the evolution of constrained states.

One-step transition

The one-step relation

$$\to \subseteq (State \times Constraints) \times (State \times Constraints)$$

describes the effect of the current instruction from a given constrained state (S, C). Most instructions are handled in the standard way, e.g. a method call instruction pushes a new activation record onto the program stack. The most interesting cases are the following ones. Suppose the current instruction is a

- *new object creation* of type T. Let $o \in \mathit{ObjectId} - O(S)$ be a fresh object identifier. Then $(S, C) \to (S', C \wedge \mathit{type}(o) = T)$ where S' is like S^{+1} except that the extensional heap $h = H_e(S)$ is replaced with

  $$\mathit{update}(\cdots \mathit{update}(h, \mathit{field}_{f_1}(o), v_1) \cdots, \mathit{field}_{f_n}(o), v_n)$$

  where $f_1, \ldots, f_n$ are the fields of T and $v_1, \ldots, v_n$ their default values, e.g. 0 for integers and `null` for references. The current instruction in S' must be a call to a constructor on o.

- *conditional branch* with a condition c and label l. If $(S, C \wedge c)$ is feasible, then $(S, C) \to (S', C \wedge c)$ where the program counter in S' is set to l. If $(S, C \wedge \neg c)$ is feasible, then $(S, C) \to (S^{+1}, C \wedge \neg c)$. Note that these cases are not exclusive and thus symbolic execution explores both branches.

- *member access* with target expression t and result location r. If $(S, C \wedge t \neq \texttt{null})$ is feasible, then the normal behavior is $(S, C) \to (S', C \wedge t \neq \texttt{null})$ where S' is like S^{+1} except that the location r holds the expression $\mathit{select}(H_e(S), \mathit{field}_f(t))$. But if $(S, C \wedge t = \texttt{null})$ is feasible as well, then there is also the exceptional transition $(S, C) \to (S'', C \wedge \mathit{type}(e) = \texttt{NullReferenceException} \wedge t = \texttt{null})$ where S'' is like S except that the current exception references a new exception object e and the program counter is advanced to the next exception handler.

- `Assume.IsTrue` with condition c. If $(S, C \wedge c)$ is feasible, then $(S, C) \to (S^{+1}, C \wedge c)$. Otherwise, there is no successor of (S, C).

- `Assert.IsTrue` with condition c. This instruction is semantically equivalent to the following code fragment:

  ```
  if (!c) throw new AssertFailedException();
  ```

Exploration

On top of the $\to$ relation, several exploration strategies like depth-first search and breadth-first search can be built. Following [9], we use a bounded depth-first search by default, which unfolds loops and recursion only a fixed number of times, using a standard set of heuristics to explore only some of the paths.

3.4 Axioms

There are two views on a PUT: It can be seen as a generator of test cases for an implementation, and as a summary, an axiom, of external behavior.

In this subsection we describe how we use uninterpreted function symbols to represent summaries, and how we generate universally quantified formulas from a PUT. These formulas can then be used by the theorem prover as rewrite rules to reason about externally observable method behavior.

Assume that we want to summarize the set of methods M of class D. We can do so by decorating a test class T_D, which contains a set of test axioms over the methods M of class D, with the attribute `[ProvidingAxioms(typeof(`D`))]`. Then other test classes will reuse these axioms when they are decorated with the attribute `[UsingAxioms(typeof(`T_D`))]`.

For method calls to M we refine the behavior of $\to$ relation. Suppose the current instruction in (S, C) is a

- *call to a method* $m \in M$, with actual arguments $\bar{x}$. Let $H_i' = m_s(H_i(S), \bar{x})$. If $(S, C \wedge m_x(H_i(S), \bar{x}) = \texttt{void})$ is feasible, then there is a normal successor with incremented program counter and intensional heap H_i'. The return value of a normal execution of m (if any) is given by the expression $m_r(H_i(S), \bar{x})$. If $(S, C \wedge m_x(H_i(S), \bar{x}) \neq \texttt{void})$ is feasible, then there is an abrupted successor with intensional heap H_i' which throws an exception of type $m_x(H_i(S), \bar{x})$ and advances to the next exception handler accordingly.

Axioms for normal behavior

We generate axiom formulas for normal behavior by exploring a test axiom method like we do for test case generation, but instead of verifying that assertions hold, we turn them into axiom formulas. More formally: We instantiate the parameters of the test axiom method with a vector $\bar{x}$ of symbolic variables. We explore the method with a modified one-step relation $\to'$, starting with a variable intensional heap h. $\to'$ is like $\to$ except that a call

`Assert.IsTrue(`c`)` in a constrained state (S, C) is treated like `Assume.IsTrue(`c`)` and in addition a new axiom formula

$$\forall h, \bar{x}.PC(C) \rightarrow c$$

is generated and conjoined with the background $BG(C)$ for further explorations. Note that both the path condition $PC(C)$ and the assertion c might refer to the variables h and $\bar{x}$.

Let's revisit the `TestAdd` axiom from Section 2.2 for an illustration of the axiom formula generation process.

```
[TestAxiom]
void TestAdd(ArrayList a, object o) {
  Assume.IsTrue(a!=null);    // 1
  int i = a.Count;           // 2
  a.Add(o);                  // 3
  Assert.IsTrue(a[i] == o);  // 4
}
```

We explore this method with the symbolic variables a, o as arguments, starting with variable intensional heap h. Figure 1 describes the resulting constraints after each statement. We use "$\cdots$" to denote that an expression didn't change from step i to $i + 1$.

Exploring `TestAdd` generates the following universally quantified formula:

$$\begin{aligned}\forall h, a, o. \quad & (a = \texttt{null} \vee \mathit{subtype}(\mathit{type}(a), ArrayList)) \\ \wedge\ & a \neq \texttt{null} \rightarrow \\ & \quad getItem_r(\\ & \quad\quad Add_s(getCount_s(h, a), a, o), \\ & \quad\quad a, \\ & \quad\quad getCount_r(h, a)) = o\end{aligned}$$

Axioms for exceptional behavior

In NUnit and VSUnit, a closed test method must not throw an exception unless it is decorated with a custom attribute of the form `[ExpectedException(typeof(`T`))]`. In this case an exception compatible with type T must be thrown by the test method.

We support this specification style for PUTs, too. However we require that if a test method has an attribute `[ExpectedException(`$\ldots$`)]`, then the expected exception must be thrown by the last call to a method in M. Further, we do not allow the presence of exception handling code in PUTs which provide axioms.

To generate axiom formulas reflecting the presence or absence of exceptions, we adapt the existing exploration. Assume that a test method with attribute `[ExpectedException(typeof(`T`))]` is being explored starting with a variable intensional heap h and symbolic variables as arguments $\bar{x}$. When we encounter the last call to a method $m \in M$ with actual arguments $\bar{y}$ in a constrained state (S, C), then we generate the axiom formula

$$\forall h, \bar{x}.PC(C) \rightarrow \mathit{subtype}(m_x(H_i(S), \bar{y}),\ T)$$

which states that an exception whose type is compatible with type T will be thrown under the path condition. For every other call to a method $m \in M$ with arguments $\bar{y}$ in a constrained state (S, C), we generate the axiom formula

$$\forall h, \bar{x}.PC(C) \rightarrow m_x(H_i(S), \bar{y}) = \texttt{void}$$

which states that $m(\bar{y})$ doesn't throw an exception when called under the path condition.

Consider the following example.

```
[TestAxiom, ExpectedException(
                typeof(ArgumentNullException))]
void TestAdd(Hashtable ht, object o) {
  Assume.IsTrue(ht!=null);
  ht.Add(null, o);
}
```

This axiom states that the `Add` method of a hash table must not be applied to a `null` key, and that an appropriate exception will be thrown otherwise. The corresponding axiom formula is

$$\begin{aligned}\forall h, ht, o. \quad & (ht = \texttt{null} \vee \mathit{subtype}(\mathit{type}(ht), Hashtable)) \\ \wedge\ & ht \neq \texttt{null} \rightarrow \\ & \quad \mathit{subtype}(Add_x(h, ht, \texttt{null}, o), \\ & \quad\quad \texttt{ArgumentNullException})\end{aligned}$$

Axioms for behavioral purity

Existing unit test frameworks like NUnit and VSUnit do not support notations to specify that a method invocation is behaviorally pure, by which we mean that it does not affect the externally observable behavior of any later method invocation (see also [3] for static techniques to check behavioral purity).

We discovered the necessity for these kinds of PUTs in the context of axiom reuse. Often, specifications would be incomplete without assertions about behavioral purity.

We support the specification of behaviorally purity by means of a special custom attribute `[ExpectedBehavioralPurity]`, which can be attached to test methods.

To generate behavioral purity axiom formulas, we again adapt the existing exploration. In more detail: Assume that a test method with attribute `[ExpectedBehavioralPurity]` is being explored starting with variable intensional heap h and symbolic variables as arguments $\bar{x}$. When an execution path through the test method ends in a constrained state (S, C), we generate the axiom formula

$$\forall h, \bar{x}.PC(C) \rightarrow H_i(S) = h$$

which states that, for all initial intensional heaps and test method arguments, the final intensional heap $H_i(S)$ can be equated with the initial intensional heap h under the path condition.

In Section 2.5, we introduced axioms for the hash table. These axioms should have included the following two purity axioms.

```
[TestClass, ProvidingAxioms(typeof(Hashtable))]
class HashtableTests {
  ...
  [TestAxiom, ExpectedBehavioralPurity]
  void ContainsIsPure(Hashtable h, object key) {
    Assume.IsTrue(h!=null && key!=null);
    bool result = h.ContainsKey(key);
  }

  [TestAxiom, ExpectedBehavioralPurity]
  void GetIsPure(Hashtable h, object key) {
    Assume.IsTrue(h!=null && key!=null);
    object result = h[key];
  }
}
```

Together, these four hash table axioms are sufficient to explore the `AddMultiplicityTest` of Section 2.5 *without* resorting to the hash table implementation.

Our approach allows us to specify behavioral purity not only for single method invocations, but also for the combined effect of a sequence of method invocations. The following example states that, if a key is not in a hash table, then adding and removing this key will leave the hash table in the same state as it was initially.

```
[TestAxiom, ExpectedBehavioralPurity]
void AddRemoveIsPure(
         Hashtable h, object key, object val) {
  Assume.IsTrue(h!=null && key!=null);
  Assume.IsTrue(!h.ContainsKey(key));
  h.Add(key, val);
  h.Remove(key);
}
```

	Path conditions	Intensional heap	Local binding
0	$(a = \texttt{null}$	$h_0 = h$	
	$\vee \mathit{type}(a) = ArrayList)$,		
1	$\ldots \wedge a \neq \texttt{null}$	$\ldots$	
2	$\ldots$	$h_1 = getCount_s(h_0, a)$	$i = getCount_r(h_0, a)$
3	$\ldots$	$h_2 = Add_s(h_1, a, o)$	$\ldots$
4	$\ldots \wedge getItem_r(h_2, a, i) = o$	$\ldots$	

Figure 1: Evolution of constraints for the axiom formula generation of the `TestAdd` method

Our implementation currently does not generate test cases for such purity axioms. See Section 5 for further discussion of how to test behavioral equivalence.

3.5 Test case generation

Each transition sequence $(S_0, C_0) \rightarrow (S_1, C_1) \rightarrow \cdots$ represents a unique execution path of the program. In this subsection, we only consider finite execution paths. We say that a path is *terminated* if it ends in a state with an empty stack of activation records.

A test case is now simply an assignment that is (together with a fixed interpretation) a solution of the last constrained state of a terminating path. By choosing one assignment per terminated execution path, we get the minimal number of test cases that cover all explored execution paths.

We say that a test case is successful if either the last state of the test case's path has no current exception and no `[ExpectedException(...)]` attribute is given, or the last state has a current exception whose type is compatible with type T of a given `[ExpectedException(`T`)]` attribute. Otherwise the test case failed, and the current exception in the last state indicates the kind of failure. It could be an assertion failure or another implicit failure.

In general, we leave it to the constraint solver to decide feasibility of constraints. If a formula is not satisfiable, the theorem prover Simplify[12] or Zap[28] might produce a counter example. In this case the verdict is clearly that the path represented by the formula is infeasible. However sometimes, our theorem provers can neither prove nor disprove the formula. In this case the verdict is inconclusive and we continue the exploration as if the path was feasible. Later, when we try to obtain concrete solutions, it might turn out that no solution exists.

But how do we get concrete assignments? For certain formulas, Zap supports model generation, i.e. the generation of assignments that fulfill the formulas. With Simplify as our constraint solver we have to provide additional domain information, similar to Korat's finitization [7]. We then use linear and binary search techniques to narrow down the space of potential solutions to a particular assignment. For instance, for symbolic variables with reference types we enumerate through the available object identifiers and the `null` value to find solutions.

Both theorem provers cannot reason about modulo and division. In order to efficiently reason about these operations, as required e.g. to explore the hash table code, we transform modulo and division by a constant into equivalent disjunctions. For example, the expression $(i/10)$ with the constraint $i \in \{0, \ldots, 29\}$ in the path condition is replaced by the expression $(i - d)$, where d is a new symbolic variable. The constraint $d \in \{0, 10, 20\}$ and the following constraint are added to the path condition.

$$\begin{array}{lllll} & (0 \leq i < 10 & \wedge & d = 0) \\ \vee & (10 \leq i < 20 & \wedge & d = 10) \\ \vee & (20 \leq i < 30 & \wedge & d = 20) \end{array}$$

If division and modulo operations are used, a finitization, here a range, must be given for the free variables involved.

4. EVALUATION

We wrote PUTs for several algorithms and collection types and generated test cases:

- Quicksort is a recursive algorithm sorting an integer array. We tested this algorithm with arrays of size 4, 5, and 6.
- Another function classifies a triangle, given by the lengths of its three sides, into one of the categories equilateral, isosceles, scalene, and invalid.
- The `ArrayList`, its enumerator, and the `Hashtable` data type were taken from the .NET Framework Version 1.1, with minor modifications to allow symbolic execution within the capabilities of the theorem provers we were using. In particular, the hash table was using bit operations to manipulate the highest bit of hash values to encode collisions. We introduced a separate boolean flag for this purpose. The array list was tested with inputs of sizes up to 10. The enumerator was tested for up to 4 elements, and the hash table for up to 2 contained elements.
- We implemented the `Bag` data type on top of `Hashtable` as outlined in this paper. We tested its axioms without reusing hash table axioms ("deep"), and with reusing ("shallow").
- The `LinkedList` and `RedBlackTree` data types were taken from an early version of the collection library of the forthcoming Spec# programming system[2], in which they are private classes used for the implementation of the `Map` data type. The assertions for the red-black tree operations mandate that the red-black tree invariant is maintained. Linked list operations were tested up to a depth of 10, red-black tree operations with up to 8 involved nodes, which include nodes created during the operations.

Figure 2 shows the results for algorithms, and Figure 3 for the data type operations.

We give the number of operations we tested, the number of PUTs we wrote, separating normal from exceptional behavior tests for data type operations. We give the number of concrete test cases that were automatically generated from the PUTs. We achieved 100% coverage of the reachable branches of the tested operations in every case. Finally, we give the time it took to generate all cases on a Pentium 4, 3.2 GHz. Memory consumption was not a concern since we performed a bounded depth-first search.

We wrote PUTs ranging from simple robustness tests, asserting only that no exception is thrown, to PUTs that relate the inputs and outputs of a method call sequence by assertions. The PUTs presented below can be seen as representative of all written PUTs.

Algorithm	Input size	PUTs	# Cases	Time
Quicksort	int[4]	2	24	0.3s
Quicksort	int[5]	2	120	1.2s
Quicksort	int[6]	2	720	8.8s
Triangle	3 sides	4	9	0.2s

Figure 2: Evaluation Results - Algorithms

Datatype	Operations	Input size	Normal PUTs	Except. PUTs	# Cases	Time
`ArrayList`	10	3	8	4	34	3.6s
`Enumerator`	4	4	4	6	67	9.8s
`Hashtable`	9	2	6	5	30	29.9s
`Bag` (deep)	3	any	3	3	20	37.2s
`Bag` (shallow)	3	any	3	3	9	2.3s
`LinkedList`	3	10	3	0	64	3.6s
`RedBlackTree`	3	8 nodes	3	0	457	427s

Figure 3: Evaluation Results - Traditional data structures

Using our prototype tool Unit Meister, we found three violations of the PUTs which we had written:

- For the `Capacity` property of the `ArrayList`, the MSDN documentation for .NET Framework Version 1.1 states:

  > "... If `Capacity` is explicitly set to zero, the common language runtime sets it to the default capacity instead. The default capacity is 16."

  We wrote the following PUT:

  ```
  [TestAxiom]
  void SetCapacityTest(ArrayList a, int i) {
    Assume.IsTrue(a!=null && i>=a.Count);
    a.Capacity = i;
    if (i==0) Assert.IsTrue(a.Capacity==16);
    else      Assert.IsTrue(a.Capacity==i);
  }
  ```

  The assertion `a.Capacity==16` fails when initially `i==0`, `a._size==0`, and `a._items.Length==0`. In this case the capacity is not set to the default capacity, but left unchanged. The documentation in Version 2.0 no longer mentions this special case.

- We found a bug in the enumerator of `ArrayList` with the following PUT describing the normal iteration behavior:[1]

  ```
  [TestAxiom]
  void IterateTest(ArrayList a) {
    Assume.IsTrue(a != null);
    IEnumerator e = a.GetEnumerator();
    for (int i = 0; i < a.Count; i++) {
      Assert.IsTrue(e.MoveNext() &&
                    e.Current == a[i]);
    }
    Assert.IsTrue(!e.MoveNext());
  }
  ```

  Symbolic execution reveals that an exception will be thrown by the `Current` property getter if, beside other constraints, `a._items[0]==a`. It turns out that in the .NET Framework Version 1.1, the array list enumerator has a field `currentElement`, which holds the array list object itself as a magic value in the special states where no current element is available, i.e. before the first call to `MoveNext`, and after `MoveNext` has returned `false`. The `Current` property getter throws an exception in these states. But since it is possible to add the array list object to itself, the `Current` property getter will throw an exception when the enumeration reaches a contained array list object. The forthcoming Version 2.0 no longer has this erroneous behavior.

- To test the static `Remove` method of the `LinkedList` class, we started by writing a robustness test:

  ```
  [TestAxiom]
  void RemoveTest(LinkedList l, object o) {
    LinkedList res = LinkedList.Remove(l, o);
  }
  ```

  Symbolic execution finds that the `Remove` method throws a `NullReferenceException` because it attempts to access n`.tail.tail` when n`.tail==null`, where n ranges over all (non-null) linked lists which do not contain o.

5. RELATED AND FUTURE WORK

The automatic generation of tests has recently received a lot of attention. Here we only try to cover those strands of research 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 with a certain model coverage.

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 [13],

[1] In fact, this axiom must be observed by every implementer of the `IList` interface.

which proposed it for testing implementations described by VDM programs. It was recently reworked by [8], 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[23] and Java PathFinder[29]. 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[7], 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. [4]. 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. One of the complications of black-box testing is to detect whether two objects on the level of the IUT are observationally equivalent (as specified by the axioms). ASTOOT tests equality of two objects of the IUT by calling a sequence of observer functions on both and comparing the respective results [14].

In parameterized unit testing we are not concerned with generating test sequences. Instead we just check whether a given property holds for the IUT (i.e. is a theorem of the implementation).

Test sequences are meanwhile also generated using symbolic evaluation of the IUT. To minimize the number of different method sequences, Symstra[30] uses heap isomorphism and state subsumption. We want to add both techniques to our framework too.

Another test goal is to test for robustness of individual methods. Usually this is done by random input data generation [10]. Recently [11] combined random input generation with constraint solving to test the robustness of individual methods, where ESC/Java [15] reports give constraint systems indicating necessary conditions for potential errors. These reports are often false positives, i.e. the constraints are unsatisfiable. DART[16] 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.

Another interesting strand is Henkel [18]'s recent work on synthesizing algebraic axioms from a given implementation. In fact, our algebraic representation of methods and of their effects follows [18]. We have extended it to not only capture normal but also exceptional behavior. Heaps in [18] are very restricted. Essentially, he uses one intensional heap per object and thus cannot describe inter-object relationships or fields updates. For synthesizing algebraic axioms, he exercises the IUT systematically while observing the relationship between the results of the individual method calls; next he generalizes the observations. Their technique works well for encapsulated abstract data types; however it does not work for synthesizing equations for designs that involve peer to peer relations like collections and their enumerators (as exemplified in the .NET library) or the subject-observer pattern. Despite these limitations, we think that this work fits well into our framework and thus we intend to adopt and extend it. If successful, it could be of tremendous help in documenting and summarizing existing libraries, such as the .NET Framework base class library.

But one should note, that there are also problems with the chosen framework of algebraic data types: for instance we model the intensional heap as the collection of all summarized objects. We saw that behavioral purity of methods or method sequence can adequately be expressed as an axiom, but it is an open question how the independence of objects, or clusters of objects, or methods over them, can elegantly be expressed in this framework. We are currently experimenting how best to define clusters by grouping together entangled types which share some state, for example the `ArrayList` and its enumerator class.

A related problem arises when an axiom describing a property of an ADT depends on properties of objects-under-test, for which summaries are not yet available. Consider for example the PUT `SetImpliesContainsAndGet` in Section 2.2. Although correct, the axiom literally generated from this PUT relates the indexer and the `ContainsKey` method only for a common object `key`, i.e. for keys that are reference-equal. But we want the axiom to apply for two arbitrary objects as long as they are related by a user-provided equality relation; in other words, the axiom itself is parameterized over the particular equality relation. We will address this problem in future work.

Symbolic execution in an object-oriented framework requires an adequate treatment of subtyping. In general, if an operation like a virtual-method call depends on an object's runtime type, the operation's execution gives rise to a non-deterministic choice over possible types and matching methods. This poses no problem if we have a closed program. Otherwise, we do not only have to generate input objects of known types, but we have to manufacture new subtypes which enable the exploration of all feasible paths; these manufactured types serve the same purpose as mock types in unit testing. This is another area we want to investigate.

Using the theory of algebraic data types opens avenues for exploring other interesting questions in testing research. One of the open questions is for instance: when do we have enough test cases — or stated differently, which test cases are missing? The typical answer is that it depends on the structural coverage of the IUT. We, however, can even give a semantic answer. A parameterized test method, i.e. axiom, is missing if observer methods (i.e. expressions that return values) cannot be reduced to independent terms (i.e., terms that don't reference the ADT-under-test). In the theory of algebraic data types this is known as testing for sufficient completeness. Our framework allows the reporting of irreducible

terms, so that the user may provide the missing specification [19]. However we do not have yet any experience with this feature.

6. CONCLUSION

We presented the concept of parameterized unit tests, a generalization of established closed unit tests. Parameterization allows the separation of two concerns: The specification of the behavior of the system, and the test cases to cover a particular implementation. Symbolic execution and constraint solving allow the automatic generation of the test cases in many cases.

We have shown how parameterized unit tests can be turned into axioms, which summarize three different aspects of the methods' behavior: state change, normal return value, and exceptional return. The axioms can be reused during symbolic execution to abstract from the implementation details of the summarized methods.

We have demonstrated the usefulness of our approach by finding bugs in the .NET Framework Library from small, simple parameterized unit tests.

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.

7. 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, *FATES 2002 workshop of CONCUR'02*, pages 105–120. INRIA Report, August 2002.

[2] M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In M. Huisman, editor, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. Conference Proceedings ICIS report NIII-R0426, University of Nijmegen, 2004.

[4] 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.

[5] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic system specification and development*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[6] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

[8] 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 *LNCS*, pages 16–32. Springer, 2004.

[9] 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.

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

[11] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ICSE*, May 2005.

[12] 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.

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

[14] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. 2002 PLDI*, pages 234–245. ACM Press, 2002.

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

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

[18] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th ECOOP*, pages 431–456, 2003.

[19] P. Jalote. Testing the completeness of specifications. *IEEE Trans. Softw. Eng.*, 15(5):526–531, 1989.

[20] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison Wesley, Boston, MA, USA, Oct. 2000.

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

[22] J. Loeckx and K. Sieber. *The Foundations of Program Verification, 2nd Edition*. Wiley, 1987.

[23] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th ASE*, pages 22–31, 2001.

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

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

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

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

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

[29] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ISSTA*, pages 97–107, 2004.

[30] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005.