Discovering Likely Method Specifications

Nikolai Tillmann[1], Feng Chen[2], and Wolfram Schulte[1]

[1] Microsoft Research, One Microsoft Way, Redmond, Washington, USA
{nikolait, schulte}@microsoft.com
[2] University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
fengchen@cs.uiuc.edu

Abstract. Software specifications are of great use for more rigorous software development. They are useful for formal verification and automated testing, and they improve program understanding. In practice, specifications often do not exist and developers write software in an ad-hoc fashion. We describe a new way to automatically infer specifications from code. Our approach infers a likely specification for any method such that the method's behavior, i.e., its effect on the state and possible result values, is summarized and expressed in terms of some other methods. We use symbolic execution to analyze and relate the behaviors of the considered methods. In our experiences, the resulting likely specifications are compact and human-understandable. They can be examined by the user, used as input to program verification systems, or as input for test generation tools for validation. We implemented the technique for .NET programs in a tool called Axiom Meister. It inferred concise specifications for base classes of the .NET platform and found flaws in the design of a new library.

1 Introduction

Specifications play an important role in software verification. In formal verification the correctness of an implementation is proved or disproved with respect to a specification. In automated testing a specification can be used for guiding test generation and checking the correctness of test executions. Most importantly specifications summarize important properties of a particular implementation on a higher abstraction level. They are necessary for program understanding, and facilitate code reviews. However, specifications often do not exist in practice, whereas code is abundant. Therefore, finding ways to obtain likely specifications from code is highly desired if we ever want to make specifications a first class artifact of software development.

Mechanical specification inference from code can only be as good as the code. A user can only expect good inferred specifications if the code serves its purpose most of the time and does not crash too often. Of course, faithfully inferred specifications would reflect flaws in the implementation. Thus, human-friendly inferred specifications can even facilitate debugging on an abstract level.

Several studies on specification inference have been carried out. The main efforts can be classified into two categories, static analysis, e.g., [16,15,14], and dynamic analysis, e.g., [13,19]. The former tries to understand the semantics of the program by analyzing its structure, i.e., treating the program as a white-box; the latter considers the implementation as a black box and infers abstract properties by observations of program runs. In

Z. Liu and J. He (Eds.): ICFEM 2006, LNCS 4260, pp. 717–736, 2006.

this article we present a new technique to infer specifications which tries to combine the strengths of both worlds. We use symbolic execution, a white box technique, to explore the behaviors of the implementation as thoroughly as possible; then we apply observational abstraction to summarize explored behaviors into compact axioms that treat the implementation as a black box.

We applied the technique to infer specifications for implementations of abstract data types (ADTs) whose operations are given as a set of *methods*, for example, the public methods of a class in C#. The technique infers a likely specification of one method, called the *modifier method*, by summarizing its behavior, e.g. its effect on the state and its result value, using other available methods, called *observer methods*. Interestingly, our technique does neither require that the modifier methods changes the state nor that observer methods do not change it.

The inferred specifications are highly abstract and human beings can review them. In many cases, they describe all behaviors of the summarized method. For example, our tool, called Axiom Meister, infers the following specification for the `Add` method of the BCL `Hashtable` class using the observer methods `ContainsKey`, the property `Count` and the indexer property `[]`.

```
void Add(object key, object value)
  requires key != null otherwise ArgumentNullException;
  requires !ContainsKey(key) otherwise ArgumentException;
  ensures ContainsKey(key);
  ensures value == this[key];
  ensures Count == old(Count) + 1;
```

Our technique obtains such a specification in three steps, illustrated in Figure 1.

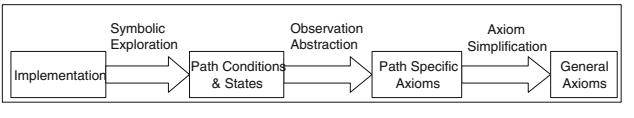

Fig. 1. Overview of the Specification Inference Process

Firstly, we symbolically execute the modifier method from an arbitrary symbolic state with arbitrary arguments. We assume single-threaded, sequential execution. Symbolic execution attempts to explore all possible execution paths. Each path is characterized by a set of constraints on the inputs called the *path condition*. The inputs include the arguments of the method as well as the initial state of the heap. The number of paths may be infinite if the method contains loops or employs recursion. Our approach selects a finite set of execution paths by unrolling loops and unfolding recursion only a limited number of times. A path may terminate normally or have an exceptional result.

Secondly, we evaluate observer methods to find an observational abstraction of the path conditions which may contain constraints referring to the private state of the implementation. Specifications must abstract from such implementation details. Observer methods are used to obtain a representation of the path conditions on a higher abstraction level. This step yields many path-specific axioms, each describing the behavior of the method under certain conditions, in terms of the observer methods.

Thirdly, we merge the collected path-specific axioms (to build comprehensive descriptions of behaviors from different cases), simplify them (to make the specification more concise), and generalize them (to eliminate concrete values inserted by loop unfolding).

The contributions of our paper are:

- We introduce a new technique for inferring formal specifications automatically. It uses symbolic execution for the exploration of a modifier method and it summarizes the results of the exploration using observer methods.
- In certain cases it can detect defective interface designs, i.e., insufficient observer methods. We show an example in Section 5 that we found when we applied our technique on code currently being developed at Microsoft.
- We can represent the inferred specifications as traditional Spec# [7] pre- and postconditions or as parameterized unit tests [26].
- We present a prototype implementation of our technique, Axiom Meister, which infers specifications for .NET and finds flaws in class designs.

The rest of this paper is organized as follows. Section 2 presents an illustrative example describing our algorithm to infer axioms, and gives an overview of symbolic execution. Section 3 describes the main steps of our technique. Section 4 discusses the heuristics we have found useful in more detail. Section 5 discusses features and limitations. Section 6 contains a brief introduction to Axiom Meister. Section 7 presents our initial experience of applying the technique on various classes. Section 8 discusses related work, and Section 9 future work.

2 Overview

We will illustrate our inference technique for an implementation of a bounded set of nonzero integers (Figure 2). Its public interface contains the methods `Add`, `IsFull`, and `Contains`. The nonzero elements of the `repr` are the elements of the set.

Here is a reasonable specification of the `Add` method in the syntax of Spec#'s pre- and postconditions [7], using `IsFull` and Contains as observer methods.

```
void Add(int x)
  requires x != 0                    otherwise ArgumentException;
  requires !Contains(x) && !IsFull() otherwise InvalidOperationException;
  ensures Contains(x);
```

Each `requires` clause specifies a precondition. Violations of preconditions cause exceptions of certain types. `requires` and `ensures` clauses are checked sequentially, e.g., `!IsFull() && !Contains(x)` will only be checked if `x!=0`. Only if all preconditions hold we can be sure that the method will not throw an exception and that the `ensures` clause's condition will hold after the method has returned.

We can also write an equivalent specification in the form of independent implications, which we call *axioms*:

$$\texttt{x==0} \Rightarrow \mathit{future}(\texttt{ArgumentException})$$
$$\texttt{x!=0} \wedge (\texttt{Contains(x)} \vee \texttt{IsFull()}) \Rightarrow \mathit{future}(\texttt{InvalidOperationException})$$
$$\texttt{x!=0} \wedge \neg\texttt{Contains(x)} \wedge \neg\texttt{IsFull()} \Rightarrow \mathit{future}(\texttt{Contains(x)})$$

```
public class Set {
  int[] repr;
  public Set(int maxSize) { repr = new int[maxSize]; }

  public void Add(int x) {
    if (x == 0) throw new ArgumentException();
    int free = -1;
    for (int i = 0; i < repr.Length; i++)
      if (repr[i] == 0) free = i;                                  // remember index
      else if (repr[i] == x) throw new InvalidOperationException(); // duplicate
    if (free != -1) repr[free] = x;                                 // success
    else throw new InvalidOperationException(); // no free slot means we are full
  }

  public bool IsFull() {
    for (int i = 0; i < repr.Length; i++) if (repr[i] == 0) return false;
    return true;
  }

  public bool Contains(int x) {
    if (x == 0) throw new ArgumentException();
    for (int i = 0; i < repr.Length; i++) if (repr[i] == x) return true;
    return false;
  }
}
```

Fig. 2. Implementation of a set

Here we used the expression *future*(_) to wrap conditions that will hold and exceptions that will be thrown when the method returns. We will later formalize such axioms.

It is easy to see that the program and the specification agree:

The `Add` method first checks if `x` is not zero, and throws an exception otherwise. Next, the method iterates through a loop, guaranteeing that the `repr` array does not contain `x` yet. The expression `!Contains(x)` checks the same condition. If the set already contains the element, `Add` throws an exception.

As part of the iteration, `Add` stores the index of a free slot in the `repr` array. After the loop, it checks if a free slot has indeed been found. `!IsFull()` checks the same condition. If the set contains no free slot, `Add` throws an exception.

Finally, `Add` stores the element in the `repr` array's free slot, so that `Contains(x)` will return `true` afterwards.

2.1 Symbolic Exploration

Our automated technique uses symbolic execution [20] to obtain an abstract representation of the behavior of the program. A detailed description of symbolic execution of object oriented programs is out of the scope of this paper, and we refer the interested reader to [17] for more discussion. Here we only briefly illustrate the process by comparing it to normal execution.

Consider symbolic execution of a method with parameters. Instead of supplying normal inputs, e.g., concrete numeric values, symbolic execution supplies symbols that represent arbitrary values. Symbolic execution proceeds like normal execution except that the computed values may be terms over the input symbols, employing interpreted functions that correspond to the machine's operations. For example, Figure 3 contains terms arising from during the execution of the `Add` method in elliptic nodes. The terms are built over the input symbols `me`, representing the implicit receiver argument, and `x`.

The terms employ the interpreted functions, including `!=`, `==`, `<`, selection of a field, and array access.

Symbolic execution records the conditions that determine the execution path. The conditions are Boolean terms over the input symbols. The path condition is the conjunction of all individual conditions along a path. For example, when symbolic execution reaches the first *if*-statement of the `Add` method, it will continue by exploring two execution paths separately. It conjoins the *if*-condition to the path condition of the *then*-path and the negated condition to the path condition of the *else*-path. Note that some branches are implicit, for example, accessing an object member might raise an exception if the object reference is `null`, and accessing an array element might fail if the index is out-of-bounds.

Not all potential execution paths are feasible. For example, after successfully accessing an object member, any subsequent member access on the same object will never fail. We use an automatic theorem prover to prune infeasible path conditions. Figure 3 shows a tree representing all feasible execution paths of `Add` up to a certain length. A path condition has a conjunct $c = v$ iff the path includes an arc labeled v from a node labeled with condition c. The figure omits arcs belonging to infeasible paths. It also omits nodes with only one outgoing arc.

The diamond nodes S2, S8, S15, S16, S23, and S24 are ends of paths that throw exceptions, and S4 and S6 represent paths terminating with errors caused by the accesses of an object member using a `null` reference. The rectangular node S14 represents a path with normal termination of the `Add` method.

2.2 Discovering Specifications from Paths

For each path, symbolic execution derives the path condition and a final program state. We could declare this knowledge as the method's specification. However, it would not be a good specification: While some of the conditions shown in Figure 3 are simple expression, e.g., `x!=0`, most expressions involve details that should be hidden from the user, like the `repr` array. And even though there are many different cases with detailed information, it is not even a complete description of the `Add` method's behavior, because symbolic exploration stopped unfolding the loop at some point. While the partial execution tree might be useful for the developer of the `Set` class, the information is simply at the wrong level of abstraction for a user of the class, who is only interested in the public interface of the ADT.

We use observational abstraction to transform the information obtained by symbolic execution into a specification, i.e., we will try to express the implementation-level conditions of the explored paths with equivalent observations that we can make on the level of the class interface. Before we discuss the general process, we will go through the steps of our technique for our example.

Consider the paths to S4 and S6 in Figure 3. They terminate with a `null` dereference error, because either `me` or `me.repr` was `null`. Symbolic execution found these paths because it started with no assumptions about the `me` argument or the values of fields. However, C# semantics preclude a call to an instance-method using a `null`-receiver, and the constructor of the `Set` class will initialize the `repr` field with a proper array. Thus, we can safely ignore the paths S4 and S6.

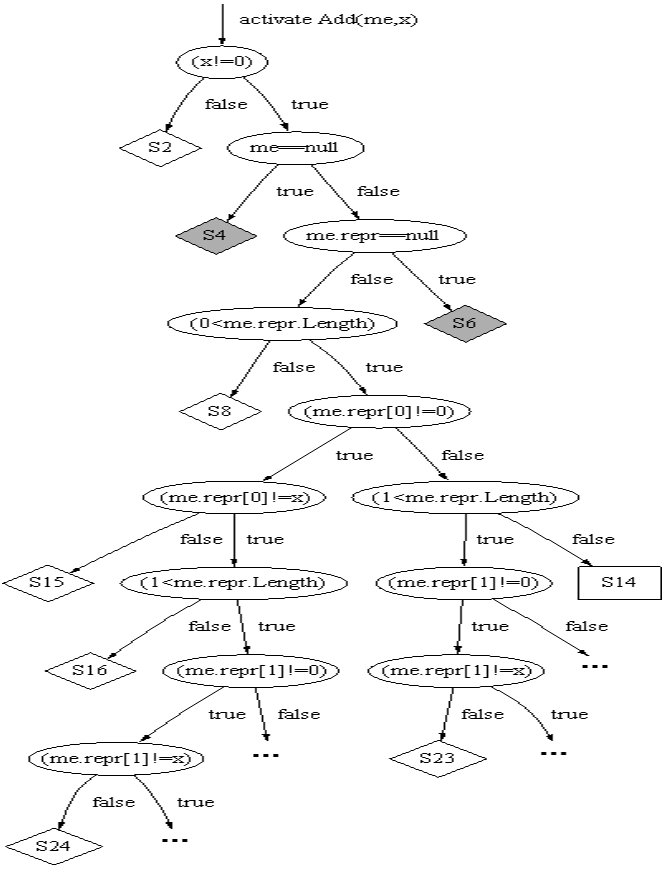

Fig. 3. Tree representation of feasible execution paths of `Set.Add` up to a certain length. See Subsection 2.1 for a detailed description.

Consider the path to S2. If `x` is zero, `Add` throws an exception. Since `x` is not private to the class, no further abstraction is necessary. We get the following precondition.

```
requires x != 0      otherwise ArgumentException;
```

Consider the paths to S15, S23 and S24. They all terminate with the same exception. In each path, the last condition establishes that `x` is equal to some element of the `repr` array. For all such `x`, `Contains(x)` clearly returns `true`. Using this characteristic behavior of `Contains`, we can summarize the paths as follows

```
requires !Contains(x) otherwise InvalidOperationException;
```

Consider the path to S8. Along the way we have `me!=null`, `me.repr!=null` and `0>=me.repr.Length`. It is easy to see that under these conditions the `IsFull` method returns `true`. Later, we will obtain this result automatically by symbolically executing `IsFull` under the constraint of path S8. The conditions along the path to S16 are

more involved; they describe the case where the `repr` array has length one and its element is nonzero. Again, `IsFull` also returns `true` under these conditions. Using this characteristic behavior of `IsFull`, we deduce:

```
requires !IsFull() otherwise InvalidOperationException;
```

We can combine the last two findings into a single `requires` clause since they have the same exception types:

```
requires !Contains(x) && !IsFull() otherwise InvalidOperationException;
```

Finally consider S14, the only normally terminating path. Its path condition implies that the `repr` array has size one and contains the value zero. Under these conditions, `IsFull` and `Contains` return `false`. (Note that when inferring preconditions, we only impose the path conditions, but do not take into account any state updates that the `Add` might perform.)

We can also deduce postconditions. Consider `Contains` under the path condition of S14 with the same arguments as `Add`, but starting with the heap that is the result of the updates performed along the path to S14. In this path the loop of `Add` finds an empty slot in the array in the first loop iteration, and then the method updates `me.repr[0]` to `x`, which will be reflected in the resulting heap. Operating on this resulting heap, `Contains(x)` returns `true`: the set now contains the added element. Consider `IsFull` under the path condition of S14 with the resulting heap. It will also return `true`, because the path condition implies that the array has length one, and in the resulting heap we have `me.repr[0]==x` where `x` is not zero according to the path condition.

After the paths we have seen so far, we are tempted to deduce that the postcondition for the normal termination of `me.Add(x)` is `Contains(x) && IsFull()`. However, when symbolic execution explores longer paths, which are not shown in Figure 3, we will quickly find another normal termination path whose path condition implies `x!=0`, with the `repr` array of size two and containing only zeros. Under these conditions, `IsFull` and `Contains` return `false` initially, the same as for S14. But for this new path, `IsFull` will remain `false` after `Add` returns since `Add` only fills up the first element of the array. Thus, the deduced postcondition will be `Contains(x) && (IsFull() || !IsFull())`, which simplifies to `Contains(x)`, in Spec#:

```
ensures Contains(x);
```

Combined, we obtain exactly the entire specification of `Add` given at the beginning. In our experiments on the .NET base class library the inferred specifications are often as concise and complete as carefully hand-written ones.

3 Technique

We fix a modifier method and a set of observer methods for this section.

3.1 Exploration of Modifier Method

As discussed in Section 2.1, we first symbolically explore a finite set of execution paths of the modifier method. Since the number of execution paths might be infinite in the presence of loops or recursion, we unroll loops und unfold recursion only a limited number of times.

3.2 Observational Abstraction

The building stones of our specifications are observations at the level of the class interface. The observations we have constructed for our example in Section 2 consisted of calls to observer methods, e.g., `Contains`, with certain arguments, e.g., `me` and `x`. In this subsection, we introduce the concepts of observer terms and observer equations which represent such observations, and we describe how we build path-specific axioms using observations.

We described in Section 2.1 how symbolic execution derives terms to represent state values and branch conditions. Consider Figure 3. While it mentions `me` explicitly, it omits another essential implicit argument: the heap. The (updated) heap is also an implicit result of each method. We view the heap as a mapping of object references to the values of their fields or array elements. Every access and update of a field or array element implicitly involves the heap. We denote the initial heap by h, and the updated heap after the method call by h'.

We extend the universe of function symbols by functions for observer methods. We write the function symbol of a method in italics. For example, the term representing the invocation `me.Contains(x)` in the initial heap h is $Contains(h, me, x)$. We write all input symbols in cursive.

The arguments are not necessarily plain input symbols, but can be terms themselves. Consider for example a method `int f(int x)`, then we can construct arbitrarily nested terms of the form $f(h, me, f(h, me, \ldots))$. We call terms over the extended universe of function symbols *observer terms*, as opposed to *ordinary terms*.

Observer equations are equations over observer terms. A *proper observer equation* does not contain heap-access subterms, e.g., field selection terms or array update terms. An example of a proper observer equation is $Contains(h, me, x) = \mathbf{true}$. In the following, we use shorthand notations for simple equations, e.g. x for $x = \mathbf{true}$, $\neg x$ for $x = \mathbf{false}$, and $x \neq y$ for $(x == y) = \mathbf{false}$.

For each explored path of the modifier method, we select a finite set of proper observer equations that is likely equivalent to the path condition. We will discuss our selection strategies in Section 4. We call those equations that do not mention the updated heap h' *(likely) preconditions*, and all other remaining equations *(likely) postconditions*. The implication from a path's preconditions to its postconditions is a *(likely) path-specific axiom*. For example, here is the axiom for path S14 in Figure 3:

$$x \neq 0 \wedge \neg \mathit{IsFull}(h, me) \wedge \neg \mathit{Contains}(h, me, x) \Rightarrow \\ \mathit{Contains}(h', me, x) \wedge \mathit{IsFull}(h', me)$$

3.3 Summarizing Axioms

For each explored path of the modifier method we compute a likely path-specific axiom. However, in most cases, the number of explored paths and thus the number of axioms is large. Obviously a human reader prefers a compact description to hundreds of such axioms. So the final step of our specification inference technique is to merge and simplify the path-specific axioms as follows:

1. Disjoin preconditions with the same postconditions
2. Simplify merged preconditions
3. Conjoin postconditions with the same preconditions
4. Simplify merged postconditions

This algorithm computes and simplifies the conjunctions of implications. The order of step 1 and 3 is not strict; changing it might result in equivalent axioms in different representations.

If a path terminates with an exception, we add a symbol representing the type of the exception to the postcondition. Section 5 discusses some exceptions to this rule.

$$
\begin{array}{lll}
x = 0 & & \Rightarrow \textit{ArgumentException} \\
x \neq 0 \wedge & IsFull(h, me) \wedge \neg Contains(h, me, x) & \Rightarrow \textit{InvalidOperationException} \\
x \neq 0 \wedge & \neg IsFull(h, me) \wedge \neg Contains(h, me, x) & \Rightarrow IsFull(h', me) \wedge Contains(h', me, x) \\
x \neq 0 \wedge & Contains(h, me, x) & \Rightarrow \textit{InvalidOperationException} \\
x \neq 0 \wedge & IsFull(h, me) \wedge \neg Contains(h, me, x) & \Rightarrow \textit{InvalidOperationException} \\
x \neq 0 \wedge & \neg IsFull(h, me) \wedge Contains(h, me, x) & \Rightarrow \textit{InvalidOperationException} \\
x \neq 0 \wedge & Contains(h, me, x) & \Rightarrow \textit{InvalidOperationException}
\end{array}
$$

Fig. 4. All Path-Specific Axioms for `Set.Add`

$$
\begin{array}{ll}
x = 0 & \Rightarrow \textit{ArgumentException} \\
x \neq 0 \wedge (IsFull(h, me) \vee Contains(h, me, x)) & \Rightarrow \textit{InvalidOperationException} \\
x \neq 0 \wedge \neg IsFull(h, me) \wedge \neg Contains(h, me, x) & \Rightarrow IsFull(h', me) \wedge Contains(h', me, x)
\end{array}
$$

Fig. 5. Merged and Simplified Axioms for `Set.Add`

Figure 4 shows all path-specific axioms of Figure 3. Figure 5 shows the equivalent merged and simplified axioms. As we discussed in Section 2.2, only when exploring longer execution paths the spurious consequence $IsFull(h', me)$ will disappear from the summarized implications in Figure 5.

```
public class Set {
  ...
  public int Count() {
    int count=0;
    for (int i = 0; i < repr.Length; i++) if (repr[i] != 0) count++;
    return count;
  }
}
```

Fig. 6. Implementation of `Set.Count`

Unrolling loops and unfolding recursion sometimes causes a series of concrete values in our axioms. Consider the extension of the bounded set class by a new observer method `Count`, given in Figure 6. The number of execution paths of the `Add` method depends on the number of loop unrollings that also determines the return value of `Count`. As a consequence, our technique infers many path-specific axioms of the following form, where α appears as a concrete number.

$$\ldots \wedge Count(h, me) = \alpha \quad \Rightarrow \quad \ldots \wedge Count(h', me) = \alpha + 1$$

Before we can merge and simplify these concrete conditions we need to generalize them into more abstract results. In this example, we are able to generalize this series of path-specific axioms by substitution:

$$\ldots \quad \Rightarrow \quad \ldots \wedge \mathit{Count}(h', me) = \mathit{Count}(h, me) + 1$$

We have also implemented the generalization of linear relations over integers.

4 Observational Abstraction Strategies

This section discusses our strategies to select proper observer equations which are likely equivalent to a given path condition. Developing these strategies is a nontrivial task and critical to the quality of inferred specifications. What we describe in this section is the product of our experience.

Since observer equations are built from observer terms, we choose the latter first.

4.1 Choosing Proper Observer Terms

A term representing an observer method call, `m(h, me, x1, ..., xn)`, involves a function symbol for the observer method, a heap, and arguments including the receiver. In the following we describe our strategies to select such proper observer term.

Choosing observer methods. Intuitively, observer methods should be *observationally pure* [8], i.e., its state changes (if any) must not be visible to a client. Interestingly, this is not a requirement for our technique since we ignore state changes performed by observer methods. However, if the given observer methods are not observationally pure, the resulting specifications might not be intuitive to users, and they might violate requirements of other tools that want to consume our inferred specifications. For example, pre- and postconditions in Spec# may not perform state updates. Automatic observational purity analysis is a non-trivial data flow problem, and it is a problem orthogonal to our specification inference. Our tool allows the user to designate any set of methods as observer methods (Figure 7). By default, it selects all property getters and query methods with suggestive names (e.g. `Get...`), which is sufficient in many cases. Since it is well known that the problem of determining a minimal basis for an axiomatic specification [12] is undecidable, we do not address this problem in our current work. In our experience, the effort of manually selecting a meaningful subset from the suggested observer methods is reasonable with the help of the GUI provided in our interactive tool which requires only a few clicks to remove or add observer methods and re-generate the specification. Our tool also allows the user to include general observer methods that test properties like $\_ = \mathbf{null}$ which have been found useful [13,19].

Choosing heaps. We are not interested to observe intermediate states during the execution of the modifier method since the client can only make observations before calling the modifier method and after the modifier method has returned. Therefore, we choose only the initial heap h or the final heap h'. The final heap represents all updates that the modifier method performs along a path.

Choosing arguments. Recall that we use symbols representing arbitrary argument values to explore the behaviors of the modifier method. A naive argument selection strategy

for an observer method is to also simply choose fresh symbols for all arguments. The following example shows when this strategy fails to detect relationships. Let `x` and `y` be two unrelated symbols, then `Contains(x)` does not provide any useful information about the behavior of `Add(y)`. As a consequence, the only symbols we use to build observer terms are the input symbols of the modifier method. And the constructed terms should be type correct.

However, for some classes this strategy is still too liberal. For example, legacy code written before generic types were available often employs parameters and results whose formal type is `object`, obscuring the assumptions and guarantees on passed values. Similarly, the presented `Set` class uses values of type `int` for two purposes: As elements of the set, e.g. in `void Add(int x)` and `bool Contains(int x)`, and to indicate cardinality, e.g. in `int Count()`.

To reduce the set of considered observer terms, we introduce the concept of *observer term groups*, or short *groups*. We associate each formal parameter and method result with a group. By default, there is one group for each type, and each parameter and result belongs to its type group. Intuitively, groups refine the type system in a way such that the program does not store a value of one group in a location of another group, even if allowed by the type system.

Lackwit [23] is a tool which infers such groups, called *extended types*, automatically for C programs. We want to implement such an analysis for .NET programs in future work. Currently, our tool allows the user to manually annotate parameters and results of methods with grouping information.

We only build group-correct observer terms: The application of an observer-method function belongs to the group of the result of the observer method, all other terms belong to the groups that are compatible with the type of the term, and the argument terms of an observer-method function must belong to the respective formal parameter group.

For example, we can assign the `int` parameters of `Add` and `Contains` to a group called `ELEM`, and the result of `Count` to a group `CARD`. When we instantiate the parameter of `Add` with x, then we will build $\mathit{Contains}(h, me, x)$ as an observer term. However, we will not consider $\mathit{Contains}(h, me, \mathit{Count}(h, me))$.

Also, our tool only builds single-nested observer terms, i.e., $f(g(x))$, and negations and equations over such terms. This has been sufficient in our experience.

4.2 Choosing Proper Observer Equations

It is easy to see how symbolic execution can reduce observer terms to ordinary terms: Just unfold the observer method functions from a given state. For example, the observer term $\mathit{Contains}(x)$ reduces to **true** when symbolically executing `Contains(x)` after `Add(x)`. The reduction is not unique if there is more than one execution path. For example, before calling `Add(x)`, we can reduce $\mathit{Contains}(x)$ to both **true** and **false**.

We fix a path p of the modifier method for the remainder of this subsection. We reduce each chosen proper observer term t relative to p as follows. We symbolically execute the observer method under the path condition of p, i.e. we only consider those paths of the observer method which are consistent with the path condition of p. Again, we only explore a limited number of execution paths. We ignore execution paths of

observer methods which terminate with an exception, and thus the reduction may also result in the empty set, in which case we omit the observer term.

For each execution path of the observer method, we further simplify the resulting term using the constraints of the path condition. For example, if the resulting term is $x = 0$ and the path condition contains $x > 0$, we reduce the result to **false**.

If all considered execution paths of the observer method yield the same reduced term, we call the resulting term the *reduced observer term of* t, written as t_R.

Given a finite set T of observer terms, we define the *basic observer equations* as $\{t = t_R : t \in T \text{ where } t_R \text{ exists}\}$. This set characterizes the path p of the modifier method by unambiguous observations. For example, the basic observer equations of S14 in Figure 3 are:

$$\{\ x = 0, \mathit{IsFull}(h, me) = \mathbf{false}, \mathit{Contains}(h, me, x) = \mathbf{false},$$
$$\mathit{Contains}(h', me, x) = \mathbf{true}, \mathit{IsFull}(h', me) = \mathbf{true}\ \}$$

However, the reductions of the observations may refer to fields or arrays in the heap, and a specification should not contain such implementation details. Consider for example a different implementation of the `Set` class where the number of added elements is tracked explicitly in a private field `count`, and the `Count` method simply returns `count`. Then the observer term $\mathit{Count}(h, me)$ reduces to the field access term $me.\mathit{count}$.

We substitute internal details by observer terms wherever possible, and construct the *completed observer equations* as follows. Initially, our completed observer equations are the basic observer equations. Then we repeat the following until the set is saturated: For two completed observer equations $t = t'$ and $u = u'$, we add $t = t'[u'/u]$ to the set of completed observer equations if the term $t'[u'/u]$ contains less heap-access subterms than t.

For example, let h' be equal to the heap for a path where `Add` returns successfully and increments the private field `count` by one, then $\mathit{Count}(h, me)$ reduces to $me.\mathit{count}$ and $\mathit{Count}(h', me)$ to $me.\mathit{count} + 1$ in the initial heap h. Then the completed observer equations will include the equation $\mathit{Count}(h', me) = \mathit{Count}(h, me) + 1$ which no longer refers to the field `count`.

We select the set of observer equations likely equivalent to p's path condition as follows: the completed observer equations less all tautologies and all equations which still refer to fields or arrays in a heap. (This way, all the remaining equations are proper observer equations.)

5 Further Discussion

Detecting insufficient observer methods. When we applied our tool to a code base that is currently under development (a refined DOM implementation [3]), our tool inferred a specification for the method `XElement.RemoveAttribute` that we did not expect.

```
void RemoveAttribute(XAttribute a)
  requires HasAttributes() && a!=null;
  ensures false;
```

This axiom is contradictory. The reason is the set of available observer methods: For some paths, `RemoveAttribute` assumes that the element contains only one attribute, then after removal, `HasAttributes` will be false. For other paths, it assumes that the element contains more than one attribute, which makes `HasAttributes` true after removal. The existing observer methods of the class `XElement` cannot distinguish these two cases. Therefore, for the same preconditions, we may reach two contradictory postconditions. This actually indicates that the class should have more observer methods. We call a set of observer methods *insufficient* if they cause our analysis to derive contradictory postconditions.

Indeed, after adding a new observer method called `AttributesCount` to the class `XElement`, we obtain the following consistent specification where `old(`e`)` denotes the value of e at the entry of the method.

```
void RemoveAttribute(XAttribute a)
  requires HasAttributes() && a!=null;
  ensures old(AttributesCount() > 1) => HasAttributes();
  ensures old(AttributesCount() < 2) => !HasAttributes();
```

This way, our tool examines if a class interface provides sufficiently many observer methods for the user to properly use the class.

Pruning unreachable states. Since we explore the modifier method from an arbitrary state, we might produce some path-specific axioms that have preconditions which are not enabled in any reachable state.

For example, for the .NET `ArrayList` implementation the number of elements in the array list is at most its capacity; a state where the capacity is negative or smaller than the number of contained elements is unreachable. Symbolic execution of a modifier like `Add` will consider all possible initial states, including unreachable states. As a consequence, we may produce specifications which describe cases that can never happen in concrete sequences of method calls. These axioms are likely correct but useless.

Ideally, the class would provide an observer method which describes when a state is reachable. Fortunately, our experiments show that this is usually not necessary. Exploration from unreachable states often results in violations of contracts with the execution environment, e.g., `null` dereferences. Since our approach assumes that the implementation is "correct," our tool prunes such error cases.

Computing the set of reachable states precisely is a hard problem. A good approximation of reachable states are states in which the class-invariant holds. If the class provides a Boolean-valued method that detects invalid program states, our tool will use it to prune invalid states.

Redundancy. Two observations might be equivalent, e.g., `IsEmpty()` is usually equivalent to `Size()==0`. While this may cause some redundancy in the generated specifications, it does not affect the soundness of the specifications. We do not provide an automatic analysis to find an expressive and minimal yet sufficient set of observer methods but leave it to the user to select an appropriate set. As we discussed in Section 4, the required effort of manually selecting observer methods has been reasonable in our experiences.

Limitations. There is an intrinsic limitation in any automatic verification technique of nontrivial programs: there cannot be an automatic theorem prover for all domains. Currently, our exploration is conservative for the symbolic exploration: if the theorem prover cannot decide a path condition's satisfiability, exploration proceeds speculatively. Therefore, it might explore infeasible paths. The consequences for the generated axioms are similar to the ones for unreachable, unpruned states.

Moreover, as mentioned, our technique considers only an exemplary subset of execution paths and observer terms. In particular, we unroll loops and recursion only a certain number of times, but the axioms in terms of the observer methods often abstract from that number, pretending that the number of unrollings is irrelevant. Without precise summaries of loops and recursion, e.g., in the form of annotated loop invariants, we cannot do better. The generalization step introduces another source of errors, since it postulates general relations from exemplary observations using a set of patterns.

While our implementation has the limitations discussed above, in our experience the generated axioms for well-designed ADTs are comprehensive, concise, sound and actually describe the implementation.

6 Implementation

We have implemented our technique in a tool called Axiom Meister. It operates on the methods given in a .NET assembly.

We built Axiom Meister on top of XRT [17], a framework for symbolic execution of .NET programs. XRT represents symbolic states as mappings of locations to terms plus a path condition over symbolic inputs. XRT can handle not only symbols for primitive values like integers, but also for objects. It interprets the instructions of a .NET method to compute a set of successor states for a given state. It uses Simplify [11] or Zap [6] as automatic theorem provers to decide if a path condition is infeasible.

Corresponding to the three steps of the inference process, Axiom Meister consists of three components: the observer generator, the summarization engine, and the simplification engine. The observer generator manages the exploration process. It creates exploration tasks for the modifier and observer methods which it hands down to the XRT framework. From the explored paths it constructs the observation equations, as discussed in Section 4.1. The simplification engine uses Maude [4].

Axiom Meister is configurable to control the execution path explosion problem: The user can control the number of loop unrollings and recursion unfoldings, and the user can control the maximum number of terminating paths that the tool considers. By default, Axiom Meister will terminate the exploration when every loop has been unrolled three times, which often achieves full branch coverage of the modifier method. So far, we had to explore at most 600 terminating paths of any modifier method to create comprehensive axioms.

Axiom Meister can output the inferred specifications as formulas, parameterized unit tests [26], or as Spec# specifications.

The user can control Axiom Meister from the command line and it has a graphical user interface (Figure 7). The user can choose the modifier method to explore, *Hashtable.Add* in this example, and a set of observer methods on the left panel. The

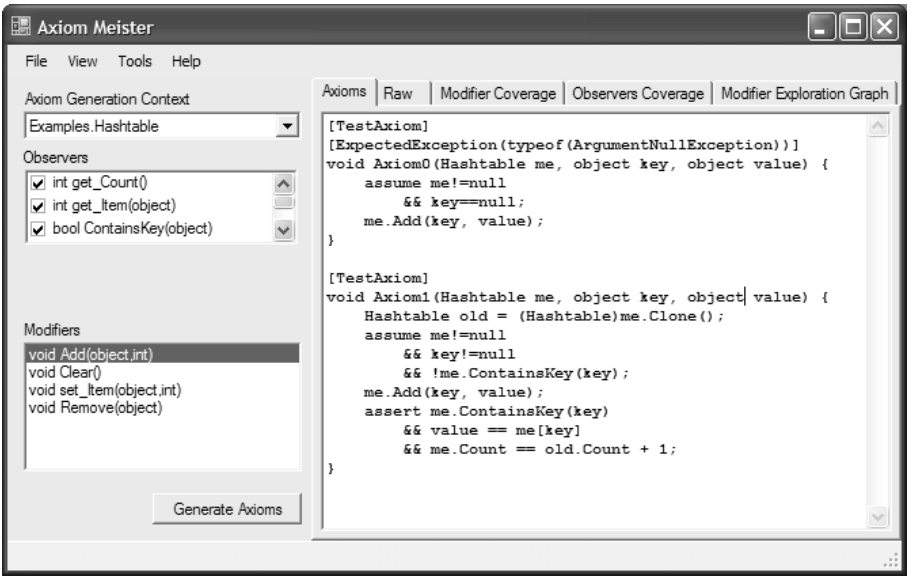

Fig. 7. Screenshot of Axiom Meister

right window shows the generated axioms, here as parameterized unit tests. It also provides views of the modifier exploration tree (Figure 3), and the code coverage of the modifier and observer methods.

7 Evaluation

We have applied Axiom Meister on a number of nontrivial implementations, including several classes of the .NET base class library (BCL), classes from the public domain, as well as classes currently under development by a Microsoft product group.

Table 1. Example Classes for Evaluating Axiom Meister

Class	Modifiers	Observers	LOC	Source
`Stack`	3	3	200	.NET BCL
`BoundedStack`	2	4	160	Other
`ArrayList`	7	6	350	.NET BCL
`LinkedList`	6	4	400	Other
`Hashtable`	5	4	600	.NET BCL
`XElement`	2	3	800	MS internal

Table 1 shows some of the investigated classes along with the numbers of the chosen modifier and observer methods. The LOC column gives the number of lines of non-whitespace, non-comment code. We took `Stack`, `ArrayList` and `Hashtable` from the BCL; `BoundedStack` is a modified version of `Stack` with a bounded size; `LinkedList` from [1] implements a double linked list with an interface similar to

`ArrayList`; `XElement` is a class of a refined DOM model [3], which is currently under development at Microsoft. We did not change the implementations with the exception of `Hashtable`: we restricted the size of its buckets array; this was necessary to improve the performance due to limitations of the theorem prover that we used.

In addition to the regular observer methods, we included a general observer method which checks if a value is `null`.

Table 2 gives the evaluation results of these examples. The first two columns show the number of explored paths and the time cost to infer specifications for multiple modifier methods of the class. Both measurements are obviously related to the limits imposed on symbolic exploration: exploration unrolls loops and recursion only up to three times. We inspected the inferred specifications by hand to collect the numbers of the last three columns. They illustrate the number of merged and simplified axioms generated, the number of sound axioms, the number of methods for which complete specifications were generated, and the percentage of methods for which full branch coverage was achieved during symbolic execution.

Table 2. Evaluation Results of Axiom Meister

Class	Paths	Time(s)	Axioms	Sound	Complete	Coverage
`Stack`	7	1.78	6	6	3	100%
`BoundedStack`	17	0.84	12	12	2	100%
`ArrayList`	142	28.78	26	26	7	100%
`LinkedList`	59	9.28	16	13	6	100%
`Hashtable`	835	276.48	14	14	5	100%
`XElement`	42	2.76	14	13	2	100%

Most BCL classes are relatively self-contained. They provide sufficient observer methods whereas new classes under development, like `XElement`, as discussed in Section 5, often do not. In these examples branch coverage was always achieved, and the generated specifications are complete, i.e., they describe all possible behaviors of the modifier method. However, some of the generated specifications are unsound. A missing class invariant causes the unsound axioms for `LinkedList`, and we discussed the unsound axioms for `XElement` in Section 5. After adding additional observer methods, we infer sound axioms only.

8 Related Work

Due to the importance of formal specifications for software development, many approaches have been proposed to automatically infer specifications. They can be roughly divided into static analysis and dynamic detection.

8.1 Static Analysis

For reverse engineering Gannod and Cheng [16] proposed to infer detailed specifications by computing the strongest postconditions. But as mentioned, pre/postconditions obtained from analyzing the implementation are usually too detailed to understand and

too specific to support program evolution. Gannod and Cheng [15] addressed this deficiency by generalizing the inferred specification, for instance by deleting conjuncts, or adding disjuncts or implications. This is similar to the merging stage of our technique. Their approach requires loop bounds and invariants, both of which must be added manually. There has been some recent progress in inferring invariants using abstract interpretation. Logozzo [22] infers loop invariants while inferring class invariants. The limitation of his approach are the available abstract domains; numerical domains are best studied. The resulting specifications are expressed in terms of the fields of classes. Our technique provides a fully automatic process. Although loops can be handled only partially, in many cases, our loop unrolling has explored enough behavior to deduce reasonable specifications.

Flanagan and Leino [14] present another lightweight verification based tool, named Houdini, to infer ESC/Java annotations from unannotated Java programs. Based on specific property patterns, Houdini conjectures a large number of possible annotations and then uses ESC/Java to verify or refuse each of them. This way it reduces the false alarms produced by ESC/Java and becomes quite scalable. But the ability of this approach is limited by the patterns used. In fact, only simple patterns are feasible, otherwise Houdini generates too many candidate annotations, and consequently it will take a long time for ESC/Java to verify complicated properties. Our technique does not depend on patterns and is able to produce complicated relationship among values.

Taghdiri [25] uses a counterexample-guided refinement process to infer over-approximate specifications for procedures called in the function being verified. In contrast to our approach, Taghdiri aims to approximate the behaviors for the procedures within the caller's context instead of inferring specifications of the procedure.

There are many other static approaches that infer some properties of programs, e.g., shape analysis [24] specifies which object graph the program computes, termination analysis decides which functions provide bounds to prove that a program terminates [10]. All these analyses are too abstract for us; we really wanted to have axioms that describe the precise input/output behavior.

8.2 Dynamic Analysis

Dynamic detection systems discover general properties of a program by learning from its execution traces.

Daikon [13] discovers Hoare-style assertions and loop invariants. It uses a set of invariant patterns and instruments a program to check them at various program points. Numerous applications use Daikon, including test generation [30] and program verification [9]. Its ability is limited by patterns which can be user-defined. We use observer methods instead: they are already part of the class may carry out complicated computations that are hard to encode as patterns, e.g., membership checking. Also, Daikon is not well-suited for automatically inferring conditional invariants. The Java front end of Daikon, Chicory [2], can make observations using pure methods. However, it only supports pure methods without arguments, which are essentially derived variables of the class state. Daikon and our technique have different goals. We focus on inferring pre- and postconditions for methods, whereas Daikon infers invariants.

Groce and Visser [18] recently integrated Daikon [13] into JavaPathFinder [27]. Their main goal is to find the cause of a counterexample produced by the model checker. Their approach compares invariants of executions that lead to errors and those of similar but correct executions. They use Daikon to infer the invariants.

Henkel and Diwan [19] have built a tool to discover algebraic specifications for interfaces of Java classes. Their specifications relate sequences of method invocations. The tool generates many terms as test cases from the class signature. It generalizes the resulting test cases to algebraic specifications. Henkel and Diwan do not support conditional specifications, which are essential for most examples we tried.

Dynamic invariant detection is often restricted by a fixed set of predefined patterns used to express constraints and the code coverage achieved by test runs. Without using patterns, our technique can often detect relationships between the modifier and observer methods from the terms over the input symbols that symbolic execution computes. We also do not need a test suite.

Xie and Notkin [29] recently avoid the problem of inferring preconditions by inferring statistical axioms. Using probabilities they infer which axiom holds how often. But of course, the probabilities are only good with reference to the test set; nevertheless, the results look promising. They use the statistical axioms to guide test generation for common and special cases.

Most of the work on specification mining involves inferring API protocols dynamically. Whaley et al. [28] describe a system to extract component interfaces as finite state machines from execution traces. Other approaches use data mining techniques. For instance Ammons et al. [5] use a learner to infer nondeterministic state machines from traces; similarly, Evans and Yang [31] built Terracotta, a tool to generate regular patterns of method invocations from observed program runs. Li et al. [21] mine the source code to infer programming rules, i.e., usage of related methods and variables, and then detect potential bugs by locating violations of these rules. All these approaches work for different kinds of specifications and our technique complements them.

9 Future Work

Although this paper focuses on examples of classes implementing ADTs, we believe that our technique can be adopted to work for cooperating classes, like collections and their iterators, or subjects and their observers. We intend to address these challenges next. Other future work includes inferring specifications for sequences of modifier methods, inferring grouping information automatically, and inferring class invariants.

Acknowledgements

We thank Wolfgang Grieskamp for many valuable discussions and for his contributions to the Exploring Runtime, XRT, which is the foundation on which we built Axiom Meister. We also thank Tao Xie, who participated in the initial discussions that shaped this work, and Michael D. Ernst for his comments on an early version of this paper. We thank Colin Campbell and Mike Barnett for proof-reading. The work of Feng Chen was conducted while being an intern at Microsoft Research.

References

1. Codeproject. http://www.codeproject.com.
2. Daikon online manual. http://pag.csail.mit.edu/daikon/download/doc/daikon.html.
3. Document object model(DOM). http://www.w3.org/DOM/.
4. Maude. http://maude.cs.uiuc.edu.
5. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.
6. T. Ball, S. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, Redmond, WA, USA, 2005.
7. 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, 2005.
8. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *Proc. 6th Workshop on Formal Techniques for Java-like Programs*, June 2004.
9. L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
10. A. R. Byron Cook, Andreas Podelski. Abstraction-refinement for termination. In *12th International Static Analysis Symposium(SAS'05)*, Sept 2005.
11. 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.
12. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.
13. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
14. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001.
15. G. C. Gannod and B. H. C. Cheng. A specification matching based approach to reverse engineering. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 389–398, Los Alamitos, CA, USA, 1999.
16. G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, pages 188–197, July 1995.
17. W. Grieskamp, N. Tillmann, and W. Schulte. XRT - Exploring Runtime for .NET - Architecture and Applications. In *SoftMC 2005: Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science, July 2005.
18. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, Oregon, May 9–10, 2003.
19. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.
20. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
21. Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, Sept 2005.

22. F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lectures Notes in Computer Science*, Jan. 2004.
23. R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 338–348, New York, NY, USA, 1997.
24. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
25. M. Taghdiri. Inferring specifications to detect errors in code. In *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, Sept 2004.
26. N. Tillmann and W. Schulte. 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,*, pages 253–262, 2005.
27. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, 2000.
28. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. the International Symposium on Software Testing and Analysis*, pages 218–228, 2002.
29. T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, November 2005.
30. T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.
31. J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proc. the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 23–28, 2004.