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 software is implemented in an ad-hoc fashion. We describe a new way to automatically infer specifications from code. Given the code of a set of methods, 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 this article we present a new technique of inferring specifications, trying 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.

The technique has been applied on inferring 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 expressing its behavior, e.g. its effect on the state and its result value, using other available methods, called *observer methods*. In this sense, we say a *modifier method* is summarized by *observer methods*. Althought it is called the modifier method, the specifications can be inferred for methods that may or may not change the state.

The inferred specifications are highly abstract and can be reviewed by users. In many cases, they completely describe the behaviors of the modifier method. Besides, they can be automatically produced as traditional pre/postconditions, ready to be used by Spec# [7] for program verification, or in the form of parameterized unit tests [26], which are equivalent universally quantified conditional axioms. For example, Figure 1 shows the inferred specifications for `HashtableAdd` in the .NET base class library in the Spec# format.

```
class Hashtable{
  void Add(object key, object value)
    requires key! = null;
    requires !this.ContainsKey(key);
    ensures this.ContainsKey(key);
    ensures value == this[key];
    ensures this.Count == old(this.Count) + 1;
  }
}
```

Fig. 1. A Pre-/postcondition Specification of the Normal (non-exceptional) Behavior for *Hashtable.Add*

The inferrence process is consisted of three steps, as illustrated in Figure 2.

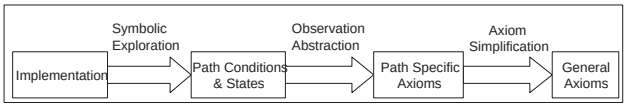

Fig. 2. Overview of the Specification Inference Process

Firstly, the modifier method is symbolically executed from an arbitrary symbolic state with arbitrary arguments. Symbolic execution attempts to explore all possible ex-

ecution 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. Presently, we assume single-threaded, sequential execution.

Secondly, observer methods are evaluated to find an observational abstraction of the path conditions. The path conditions may contain constraints over the heap in which the private fields of the observed object are stored. 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, the collected path-specific axioms are merged (to build comprehensive descriptions of behaviors from different cases), simplified (to make the specification more concise) and generalized (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, e.g., insufficient observer methods. We show two examples in Section 5 and Section 7 about the flaws found for .Net base classes during our experiment.
- We can represent the inferred specifications as traditional Spec# pre/postconditions or as parameterized unit tests.
- 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 the .NET base class libraries. Section 8 presents related work. Section 9 discusses future work.

2 Overview

We will illustrate our inference technique for an implementation of a bounded set of nonzero integers (Figure 3). Its public interface contains the methods `Add`, `IsFull`, and `Contains`. The elements of the set are stored in the array `repr`. An element in the array is zero if it has not yet been assigned an element of the set.

Here is a reasonable specification of the `Add` method using the syntax of Spec#'s pre- and postconditions [7].

```
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. 3. Implementation of a set

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

Each `requires` clause specifies a precondition. If the precondition is violated, an exception of a certain type is thrown. The `requires` clauses are checked sequentially, e.g., `!IsFull() && !Contains(x)` will only be checked if `x!=0`, and so on. Only if all preconditions hold it is guaranteed that the method will not throw an exception and that the condition of the `ensures` clause will hold after the method has returned.

Instead of a sequential Spec# specification we can also write an equivalent specification in the form of independent implications, which we call *axioms*:

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

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 `x` is not stored in the `repr` array yet. The expression `!Contains(x)` checks the same condition. If the element is already contained, an exception is thrown.

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 is full, i.e., no free slot can be found, an exception is thrown.

Finally, the element is assigned to the free slot in the `repr` array, 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 operations of the machine. For example, Figure 4 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 decide which execution path is taken. 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. The *if*-condition is conjoined to the path condition for the *then*-path and the negated condition to the path condition of the *else*-path. 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 execution paths are feasible. For example, when the same object reference is used to access a member twice, then the second access will never fail. We use an automatic theorem prover to prune infeasible path conditions. Figure 4 shows a tree representing all feasible execution paths of `Add` up to a certain length. The elliptic nodes contain the branch conditions encountered. When the path from a node with condition c along an arc labeled with `true` is taken, c is conjoined to the path condition; when the arc labeled `false` is taken, $\neg c$ is conjoined. Arcs belonging to infeasible paths are omitted. Nodes with only one outgoing arc are omitted as well.

The conditions of the form `_==null` arise from implicit checks performed when accessing object members or array elements. 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 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, we know the correspondence between path conditions and the effect of the execution (i.e. the updates on the heap and the result value computed from the

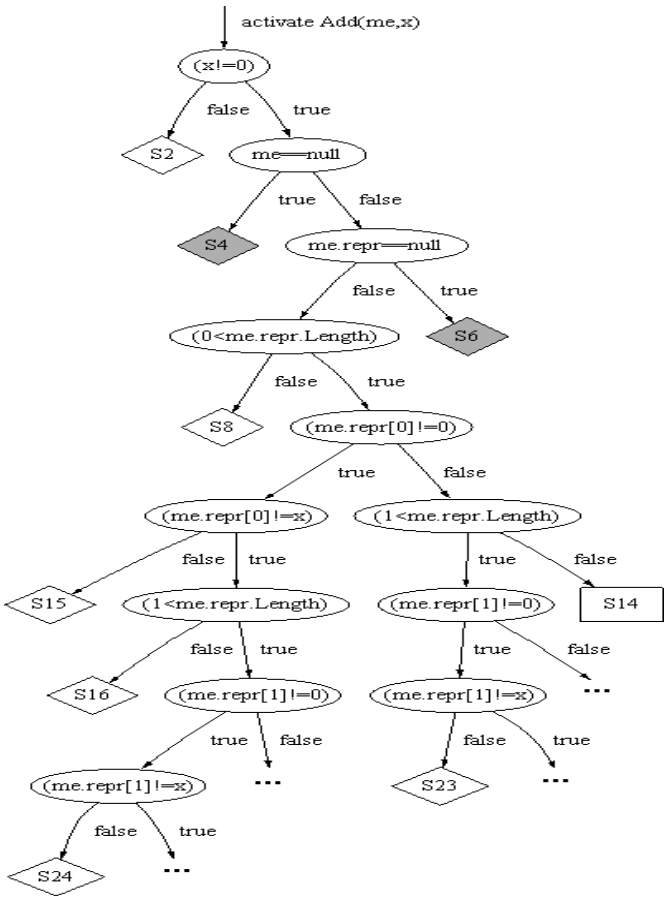

Fig. 4. Tree representation of feasible execution paths of `Set.Add` up to a certain length. See Subsection 2.1 for a detailed description.

inputs along the path). We could declare this knowledge to be the method's specification. However, there are several problems with such detailed description: While some of the conditions shown in Figure 4 are simple expression, e.g., `x!=0`, most are expressions involving details that should be hidden from the user, like the `repr` array. And even though many different cases with detailed information have been obtained, it is not even a complete description of the behavior of the `Add` method, 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 cover the implementation-level con-

ditions of the explored paths with observations that can be made on the level of the 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 4. 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 the fields on the heap. 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.

Consider the path to S2. If `x` is zero the method will terminate with an exception. Since `x` is visible outside of the class, no further abstraction is necessary, and we can write this (partial) specification as follows using Spec# syntax:

```
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. Under this condition, the symbolic execution of `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 establish 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 we only impose the path conditions, but do not take into account any heap updates that might be performed along this path for inferring the pre-conditions.)

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, the `Contains` method returns `true`: the added element is now contained. 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 4, we will quickly find another normal termination path, whose path condition implies that `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 can deduce the complete desirable specification of `Add` given at the beginning. Actually, during our experiments on the .NET base class library, the inferred specifications are often as concise and complete as carefully hand-written ones.

3 Technique

In this section, we assume that a designated modifier method and a set of designated observer methods are given.

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 in our example consisted of calls to observer methods, e.g., `Contains`, with certain arguments, e.g., `me` and `x`, where `me` is used for the implicit receiver argument.

While Figure 4 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. The heap is implicitly involved in every access and update of a field or array element. In the remainder of this paper, we will use the input symbol h for the heap. We will write all other input symbols in cursive as well.

We extend the universe of function symbols, and as a convention, the function symbol of a method will be written in cursive. For example, a term representing the invocation `me.Contains(x)` with a particular heap is $\mathit{Contains}(h, me, x)$.

The arguments are not necessarily plain input symbols, but can be terms themselves. Consider for example a class `Hashtable` which associates keys with values and provides a lookup method `get_Item`. Then we can construct arbitrarily nested terms of the form $getItem(h, me, getItem(h, me, \ldots))$. We call terms over the extended universe of function symbols *observer terms*.

Observer equations are equations over observer terms. A *proper observer equation* does not contain any heap-access subterms and does not refer to any heap but the initial heap, denoted h, and the updated heap after the method, denoted h'. An example of a proper observer equation is $getItem(h, me, x) = \mathbf{null}$, which means that `me` does not contains key `x` initially. We also use shorthand notations for simple equations, e.g. x for $x = \mathbf{true}$, $\neg x$ for $x = \mathbf{false}$, and $x \neq y$ for $(x = y) = \mathbf{false}$.

The set of observer terms and therefore the set of observer equations is infinite. However, we can only consider finite sets in our analysis. The strategies to select path-specific proper observer equations will be in Section 4. Here, we assume that an oracle provides a finite set of proper observer equations for each considered path of the modifier method. For each path, we call those equations that do not mention the updated heap h' *(likely) preconditions*, and all other remaining equations *(likely) postconditions*. The implication from the preconditions to the postconditions is the *(likely) path-specific axiom*. For example, here is the axiom for path S14 in Figure 4:

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

3.3 Summarizing Axioms

For each chosen path of the modifier method we compute a likely path-specific axiom, which is compact and expressive enough. However, in most cases, a large number of paths are explored for a modifier method and thus many path-specific axioms are computed. It is obviously that a human reader prefers a compact description to hundreds of such axioms. So the final and important step of our specification inference technique is to merge and simplify the path-specific axioms, which is accomplished 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 and can be changed to get 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.

Figure 5 shows all path-specific axioms of Figure 4. Figure 6 shows the equivalent merged and simplified axioms. Their meanings have been discussed in Section 2.2.

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

Fig. 5. 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. 6. Merged and Simplified Axioms for `Set.Add`

Unrolling loops and unfolding recursion sometimes causes extra difficulities of merging and simplification, because of concrete values inserted in the axioms. Consider for example that we extend the bounded set class with a new observer method `Count`, given in Figure 7. The number of execution paths of the `Add` method depends on the number of loop unrollings, which 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.

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

These concrete conditions cannot be merged and simplified, and we need to first generalize them into more abstract results. In this example, we are able to generalize this series of path-specific axioms by substitution:

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

We have also implemented the generalization of linear relations over integers. After successful generalization, the specification can be further merged and simplified.

4 Observational Abstraction Strategies

This section discusses our strategies to choose appropriate observer equations and terms. Developing these strategies is a nontrivial task and critical to the quality of inferred specificaitons. What we describe in this section is the product of our experience.

4.1 Choosing Observer Terms

A term representing an observer method call, `m(h, me,` x_1`, ...,` x_n`)`, involves a function symbol for the observer method, a heap, and arguments including the receiver. All of these must be fixed to construct an 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,

```
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. 7. Implementation of `Set.Count`

this is not a requirement for our technique since we ignore state changes performed by designated 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 post-conditions 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 designated any set of methods as observer methods (Figure 8). By default, it selects all property getters and query methods with suggestive names (e.g. `Get...`), which are enough for 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 GUI provided in our interactive tool, which requires only a few clicks to remove/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 do *not* want 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 original heap, identified by the symbol h, or the final heap, identified by h'. The final heap represents all updates that the modifier method performs along a certain path.

Choosing arguments. Recall that arbitry arguments are used to explore the behaviors of the modifier method. A naive argument selection strategy is to also simply choose fresh symbols for all arguments. However, it is difficult, if not impossible, to relate symbolic execution of the observer method with fresh symbols to any execution path of the modifier method. Consider for example two unrelated symbols `x` and `y`, then `Contains(x)` does not provide any useful information about the behavior of `Add(y)`. As a consequence, the only variable 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` classes uses values of type `int` for two purposes: As elements of the set, e.g. in `Add(int)` and `Contains(int)`, 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] calls these groups *extended types*, and describes a way to infer them automatically. While we did not implement automatic group inference, our tool allows the user to 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 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 $Contains(h, me, x)$ as an observer term. However, we will not consider $Contains(h, me, Count(h, me))$.

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

4.2 Choosing Proper Observer Equations

It is easy to see how observer terms can be reduced to ordinary terms by symbolic execution: Just unfold the observer method functions from a given state. For example, the observer term $Contains(x)$ reduces to $\mathbf{true}$ 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 $Contains(x)$ to both $\mathbf{true}$ and $\mathbf{false}$.

An observer term t is reduced relative to a path p of the modifier method. We fix p for the remainder of this subsection. As we discussed before, we only consider observer terms which refer to the initial heap h or the final heap h'. In this subsection, we equate h' with the particular final heap as it was updated along path p. The updated heap is usually represented by a term consisting of a chain of updates of fields and arrays, rooted in the original heap h. Then, 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 consider 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 the observer term will be omitted.

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 $\mathbf{false}$.

If all considered execution paths of the modifier 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 4 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 such observations over the internal state of the ADT should not be part of a specification. Consider for example a different implementation of the `Set` class and the `Add` and `Count` methods where the number of contained elements is tracked explicitly in private field `count` of the class. Then, the observer term $\mathit{Count}(h, me)$ will be reduced 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 $t'[u'/u]$ contains less heap-access subterms than t.

For example, let h' be equal to the heap for a path where `Add` returns successfully, and let $\mathit{Count}(h', me)$ reduce to $me.\mathit{count} + 1$ in the original 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 finally select those 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. In the previous section we mainly discussed strategies to *reduce* the number of observer terms in order to reduce the complexity of specification inference and to achieve concise specifications. Having too few observer methods is problematic as well, because they may lead to unsound specifications.

Consider the following example, which we found when we applied our tool to a code base that is currently under development (a refined DOM implementation [3]). We ran into an inferred specification for the method `XElement.RemoveAttribute` that we did not expect.

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

Obviously, this axiom is corrupted. The reason is the provided set of observer methods: For some paths, `RemoveAttribute` assumes that the element contains only one attribute, then after removal, `HasAttributes` will be false, while 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 a missing observer method in the design of the class. If contradictory postconditions can be reached for a set of observer methods, we say this set is *insufficient*.

Adding an additional observation method (`AttributesCount`) to the `XElement` class, we immediately obtain the following consistent specification axioms, 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 can be used to examine if the class interface provides sufficient observer methods for the user to properly use the class.

Pruning unreachable states.. Since we explore the modifier method from a fresh, unconstrainted 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. We do not provide an automatic analysis to find an expressive and minimal yet sufficient set of observer methods. This may cause some redundancy in the generated specifications. For example, `IsEmpty()` is usually equivalent to `Size()==0`. However, redundancy does not affect soundness of the specifications. In fact, sometimes redundant observer methods can even help in program understanding because they may describe the same behavior in different ways.

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 satisfiability of a path condition cannot be decided, symbolic execution proceeds speculatively. Therefore, infeasible paths might be explored. 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 timers, but the axioms in terms of the observer methods often abstract from that number, pretending that the number of loop unrollings is irrelevant. But 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.

Axiom Meister is built 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 will be considered. 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. 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. More details about the internal representation of Axiom Meisters axiom representation, how the state is represented symbolically and how the theorem prover is used can be found in [26].

Axiom Meister can be controlled from the command line and it has a graphical user interface (Figure 8). 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 generated axioms are then shown in the right window. It also provides views of the modifier exploration tree (Figure 4), and the code coverage of the modifier and observer methods.

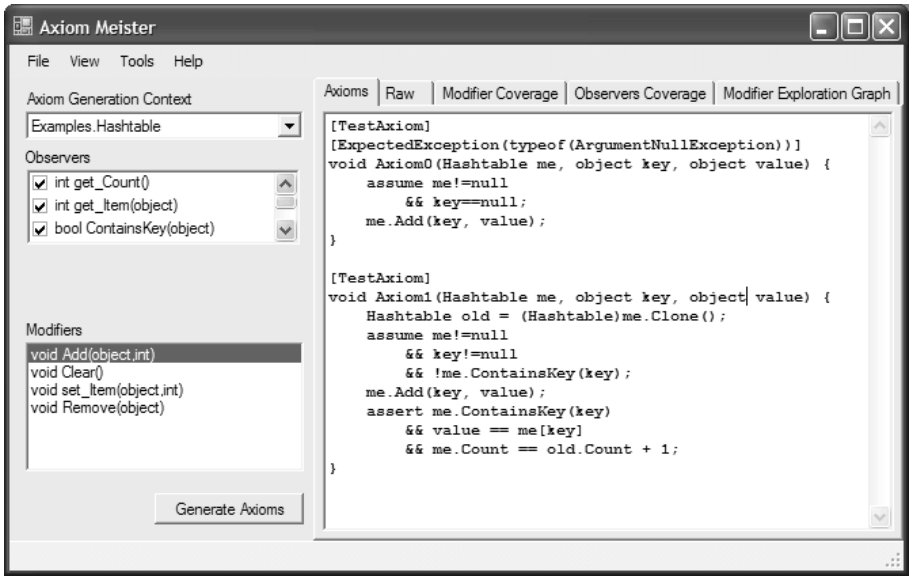

Fig. 8. Screenshot of Axiom Meister

7 Evaluation

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

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. `Stack`, `ArrayList` and `Hashtable` are taken from the BCL; `BoundedStack` is a modified version of `Stack` with a bounded size; `LinkedList` implements a double linked list with an interface similar to `ArrayList` and is taken from [1]; `XElement` is a class of a refined DOM model [3], which is currently under development. All implementations are unchanged 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 used theorem prover.

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. Example Classes for Evaluating Axiom Meister

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 is set to terminate when every loop is unrolled three times. The last three columns 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.

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%

Table 2. Evaluation Results of Axiom Meister

We inspected the inferred specifications by hand to collect the numbers of the last three columns.

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. The unsound axioms for `LinkedList` are caused by a missing class invariant, and the unsound axioms for `XElement` were discussed in Section 5. After adding additional observer methods, we infer sound axioms only.

In our experience, the inferred specifications are concise and highly readable. For example, our tool 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;
```

In another example, our tool relates the `Count` and `Capacity` properties of the `Add` method of BCL's `ArrayList` class as follows. The jump in `Capacity` from zero to four could be considered an implementation detail, and the `Capacity` property leaks out this detail. In this view, the generated specification indicates that the class provides too many observer methods.

```
void Add(object value)
  ensures old(Capacity == 0) ==> Capacity == 4;
  ensures Count == old(Count) + 1;
```

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] proposed 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, the false alarms produced by ESC/Java can be reduced and Houdini becomes quite scalable. But the ability of this approach is limited by the patterns used. In fact, only simple patterns are feasible, otherwise too many candidate annotations will be generated, 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 can be used as 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 of programs. It uses a set of invariant patterns and instruments the program to check these patterns at various program points. Daikon has been used for numerous applications, including test generation [30] and program verification [9]. Its ability is limited by the given patterns, which can be user-defined. We use observer methods instead: they are already part of the class, and they 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], provides an option to make observations on the execution using pure methods. However, it only supports pure methods without arguments, which are essentially derived variables of the class state. Daikon aims at a different goal than our technique. We focus on inferring pre/postconditions for methods, whereas Daikon infers invariants.

Groce and Visser [18] recently integrated Daikon [13] into JavaPathFinder [27]. The main purpose of their work is to find the cause of a counterexample produced by the model checker. This is achieved by comparing invariants of executions that lead to errors and those of similar but correct executions. The invariants are inferred using Daikon.

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. The results of these tests are generalized to algebraic specifications. Henkel and Diwan do not support conditional specifications, which are needed for most examples we tried.

Dynamic invariant detection is often restricted by two facts: (1) the predefined patterns used to express constraints and (2) code coverage achieved by test runs. Our technique does not use fixed patterns; instead symbolic exploration builds up terms that can express arbitrary relationships, such as non-linear integer expressions; as long as we have enough observations we have no problem summarizing them. 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 is targeted at 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 runs of the program. Li et al. [21] apply data mining in the source code to infer programming rules, i.e., usage of related methods and variables, and then detect potential bugs by locating the violation 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 the proposed technique can be adopted to work for cooperating classes, like iterators and their collections, 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 using a information-flow analysis, 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 Axiom Meister is built. 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.