DySy: Dynamic Symbolic Execution for Invariant Inference

Christoph Csallner
College of Computing
Georgia Tech
Atlanta, GA 30332, USA
csallner@cc.gatech.edu

Nikolai Tillmann
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
nikolait@microsoft.com

Yannis Smaragdakis
Computer Science Dept.
University of Oregon
Eugene, OR 97403, USA
yannis@cs.uoregon.edu

ABSTRACT

Dynamically discovering likely program invariants from concrete test executions has emerged as a highly promising software engineering technique. Dynamic invariant inference has the advantage of succinctly summarizing both "expected" program inputs and the subset of program behaviors that is normal under those inputs. In this paper, we introduce a technique that can drastically increase the relevance of inferred invariants, or reduce the size of the test suite required to obtain good invariants. Instead of falsifying invariants produced by pre-set patterns, we determine likely program invariants by combining the concrete execution of actual test cases with a simultaneous *symbolic* execution of the same tests. The symbolic execution produces abstract conditions over program variables that the concrete tests satisfy during their execution. In this way, we obtain the benefits of dynamic inference tools like Daikon: the inferred invariants correspond to the observed program behaviors. At the same time, however, our inferred invariants are much more suited to the program at hand than Daikon's hard-coded invariant patterns. The symbolic invariants are literally derived from the program text itself, with appropriate value substitutions as dictated by symbolic execution.

We implemented our technique in the DySy tool, which utilizes a powerful symbolic execution and simplification engine. The results confirm the benefits of our approach. In Daikon's prime example benchmark, we infer the majority of the interesting Daikon invariants, while eliminating invariants that a human user is likely to consider irrelevant.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants*

General Terms

Design,Languages

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION AND MOTIVATION

Dynamic invariant inference was introduced less than a decade ago, pioneered by the Daikon tool [8, 9, 29], and has garnered significant attention in the software engineering community. With the help of a test suite that exercises the functionality of an application, an invariant inference system observes program properties that hold at pre-selected program points (typically method entries and exits). The outcome of the system is a collection of such properties, postulated as object state invariants, method preconditions, or method postconditions (collectively called "*invariants*" in the following). The properties have no formal assurance that they are correct, but they do match the observed program executions and they are produced only when there is some statistical confidence that their occurrence is not accidental. A crucial aspect of the dynamic invariant inference process is that the invariants produced do not reflect only the behavior of the program, but also the assumptions and expectations of the test suite. This makes the approach doubly useful for software engineering purposes, by introducing the usage context of an application.

So far, dynamic invariant inference systems have had a pre-set collection of invariant templates, which get instantiated for program variables to produce the candidate invariants under examination. The user can expand the collection by adding more templates, but the number of possible instantiations for all combinations of program variables grows prohibitively fast. Therefore, dynamic invariant inference systems typically perform best by concentrating on a small set of simple candidate invariants. Even so, for a tool like Daikon or DIDUCE [18] to produce invariants that match the understanding of a human programmer, an extensive test suite that thoroughly exercises the application is necessary. Furthermore, it is likely that the inference process will also produce several invariants that are either irrelevant or false (i.e., hold accidentally).

In this paper we propose a *dynamic symbolic execution* technique to drastically improve the quality of inferred invariants (i.e., the percentage of relevant invariants) or the ease of obtaining them (i.e., the number of test cases required to disqualify irrelevant invariants).[1] In dynamic symbolic execution, we execute test cases, just like a traditional dynamic invariant inference tool, but simultaneously also perform a symbolic execution of the program. The symbolic execution results in the program's branch conditions being

[1]The benefit of our approach can be viewed along either axis. Holding one metric constant results in improving the other.

```
int testme(int x, int y) {
  int prod = x*y;
  if (prod < 0)
    throw new ArgumentException();
  if (x < y) {     // swap them
    int tmp = x;
    x = y;
    y = tmp;
  }
  int sqry = y*y;
  return prod*prod - sqry*sqry;
}
```

Figure 1: An example method whose invariants we want to infer.

collected in an expression, called the *path condition* in the symbolic execution literature. The path condition is always expressed in terms of the program inputs. It gets refined while the test execution takes place, and symbolic values of the program variables are being updated. At the end of execution of all tests, the overall path condition corresponds to the precondition of the program entity under examination. Symbolic values of externally observed variables provide the dynamically inferred postconditions, and symbolic conditions that are preconditions and postconditions for all methods of a class become the class state invariants.

For a demonstration of our technique, consider the method of Figure 1. (The example is artificial but is designed to illustrate several points that we make throughout the paper.) Appropriate unit tests for the method will probably exercise both the case "`x < y`" and its complement, but are unlikely to exercise the code producing an exception, as this directly signifies illegal arguments. Consider the outcome of executing the code for input values `x` smaller than `y` (e.g., `x == 2`, `y == 5`), while also performing the execution in a symbolic domain with symbolic values `x` and `y` (we overload the variable names to also denote the respective symbolic values designating the original inputs). The first symbolic condition that we observe is "`x*y >= 0`": The branch of the first `if` is not taken, and local variable `prod` has the value `x*y` in the symbolic domain. The symbolic execution also accumulates the condition "`x < y`" from the second `if` statement. At the end of execution the symbolic value of the returned expression is "`y*x*y*x - x*x*x*x`". Note that this expression integrates the swapping of the original `x` and `y` values.

If we repeat this process for more test inputs (also exercising the other valid path of the method) and collect together the symbolic conditions, then our approach yields:

- A precondition `x*y > = 0` for the method.
- A postcondition `\result == (((x < y) -> y*x*y*x - x*x*x*x) else -> (x*y*x*y - y*y*y*y))`.

(Our example syntax is a variation of JML [23]: we introduce an if-else-like construct for conciseness. Our tool's output syntax is different but equivalent.) This captures the method's behavior quite accurately, while ensuring that the only symbolic conditions considered are those consistent with actual executions of the test suite. Thus the approach is *symbolic*, but at the same time *dynamic*: the symbolic execution is guided by actual program behavior on test inputs. Note that the inferred invariants are not postulated externally, but instead discovered directly from the program's symbolic execution. This approach directly addresses many of the shortcomings of prior dynamic invariant inference tools (with Daikon used as the foremost reference point). For this example, Daikon-inferred preconditions and postconditions are exclusively of the form "*var* `>= 0`" or "*var* `== 0`", and are often encoding arbitrary artifacts of the test suite, unless a very thorough test plan exercises many possible combinations with respect to zero (e.g., `x`, `y` both negative, both positive, one/both zero, etc.).

Overall, our work makes the following contributions:

- We introduce the idea of using dynamic symbolic execution for invariant inference. We believe that our approach represents the future of dynamic invariant inference tools, as it replaces a blind search for possible invariants with a well-founded derivation of such invariants from the program's conditions and side-effects.
- We implemented our approach in the invariant inference tool *DySy*, built on top of the Pex framework for instrumentation and symbolic execution of .NET programs. We discuss the heuristics used by DySy in order to simplify symbolic conditions (e.g., our abstraction heuristics for dealing with loops). DySy represents well the benefits of the proposed technique. For instance, DySy's symbolic approach can infer invariants, such as *purity* (absence of side-effects), that are too deep for traditional dynamic tools. In contrast, prior dynamic invariant inference tools (which only observe after-the-fact effects) typically can establish purity only in very limited settings, as they would need to observe the entire reachable heap.
- We evaluate DySy in direct comparison with Daikon, in order to showcase the tradeoffs of the approach. For the `StackAr` benchmark (hand-translated into C#), which has been thoroughly investigated in the Daikon literature [10], DySy infers 24 of the 27 interesting invariants (as independently inferred by a human user), while eliminating Daikon's multiple irrelevant or accidental invariants.

The rest of this paper begins with a brief discussion of what our work is *not* (Section 2), and continues with some background on dynamic invariant inference and symbolic execution (Section 3) before detailing the technical aspects of our approach and tool (Section 4) and presenting our evaluation (Section 5). Related work (Section 6) and our conclusions (Section 7) follow.

2. POSITIONING

Our approach is a combination of symbolic execution with dynamic testing. As such, it has commonalities with multiple other approaches in the research literature. To avoid early misunderstandings, we next outline a few techniques that may at first seem similar to our approach but are deeply different.

- Our dynamic symbolic execution is not equivalent to lifting conditions from the program text (e.g., conditions in `if` statements or in `while` loops) and postu-

lating them as likely invariants. (Several prior analysis tools do this—e.g., Liblit's statistical bug isolation approach [25] and Daikon's `CreateSpinfo` supporting utility.) For instance, notice how the precondition of our example in the Introduction (`x*y > = 0`) does not appear anywhere in the program text. Instead, program conditions are changed during the course of symbolic execution: local variable bindings are replaced with their symbolic values, and assignments update the symbolic values held by variables, thus affecting the path condition.

- Our approach is not *invariant inference through static techniques* (e.g., using abstract interpretation [26], or symbolic execution [32]). Inferring invariants through static analysis is certainly a related and valuable technique, but it is missing the dynamic aspect of our work, as it takes into account only the program text and not the behavior of its test suite. Specifically, our dynamic symbolic execution uses the test suite as a way to discover properties that users of the code are aware of. This is highly valuable in practice, as invariant inference tools are often used to "read the programmer's mind" and discover the interesting parameter space of a method (e.g., for testing [5]).
- Our approach is not *concolic execution* (as in tools like Dart [17], Cute [30], or Parasoft's original "dynamic symbolic execution" patent [21]). Although we do a concrete execution of test cases in parallel with a symbolic one, we do not use the symbolic execution to produce more values in order to influence the path taken by concrete executions. Our technique follows precisely the concrete program paths that the original test suite induces.

3. BACKGROUND

We next present some background on dynamic invariant inference and on symbolic execution, emphasizing the features of both that are particularly pertinent to our later discussion.

3.1 Dynamic Invariant Inference: Daikon

Dynamic invariant inference is exemplified by (and often even identified with) the Daikon tool [8, 9, 27, 29]—the first and most mature representative of the approach, with the widest use in further applications (e.g., [2, 6, 7, 11, 22, 36]).

Daikon tracks a program's variables during execution and generalizes the observed behavior to invariants—preconditions, postconditions, and class invariants. Daikon instruments the program, executes it (for example, on an existing test suite or during production use), and analyzes the produced execution traces. At each method entry and exit, Daikon instantiates some three dozen invariant templates, including unary, binary, and ternary relations over scalars, and relations over arrays. Example relations include comparison's of a variable with a constant value (`x = a`, or `x > 0`), linear relationships (`y == a*x + b`), ordering (`x <= y`), membership and sortedness, etc. Users can extend the invariant templates with application-specific or domain-specific properties. The number of candidate invariants grows combinatorially, however. For each invariant template, Daikon tries several combinations of method parameters, method results, and object state. For example, it might propose that some method `m` never returns `null`, or that its first argument is always larger than its second. Daikon subsequently disqualifies invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned `null` and it will therefore ignore the above invariant. So Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants might hold in all other executions as well. Daikon can annotate the testee's source code with the inferred invariants as JML annotations [23].

3.2 Symbolic Execution

Symbolic execution [20] is a technique for using a program's code to derive a general representation of its behavior, by simulating execution with some values being unknown. Specifically, symbolic execution replaces the concrete inputs of a program unit (typically, a method) with symbolic values, and simulates the execution of the program so that all variables hold symbolic expressions over the input symbols, instead of values. For symbolic execution to "simulate" regular, concrete execution, its semantics must correctly generalize that of concrete execution. The key property is commutativity: performing symbolic execution and instantiating its output state with concrete values must yield the same result as instantiating the initial symbolic state with the same concrete values and performing concrete execution.

A concept of symbolic execution that is particularly important for our work is that of a *path condition*, defined as "the accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path" [20]. Thus, a path condition can be seen as a precondition for a program path, which is exactly the way we use it in our work.

Generally, the greatest challenge of symbolic execution is to reason about symbolic program properties. For instance, in traditional symbolic execution, when accumulating predicates in the path condition, it is important to recognize when the path condition becomes unsatisfiable by the addition of an extra predicate—i.e., when the existing path condition contradicts a program branch. To do so, a symbolic reasoning engine (typically an automatic theorem prover) is employed. In our approach, we do not need to recognize infeasible program paths, as the concrete execution guarantees that the paths we are examining are feasible. Nevertheless, we need similar automatic reasoning power in order to simplify path conditions and symbolic expressions and present them to the user as program invariants.

4. DYNAMIC SYMBOLIC EXECUTION FOR INVARIANT INFERENCE

We next discuss the general elements of our approach, as well as the technical specifics of our DySy tool, and the abstraction heuristics we employ for handling loops.

4.1 Overview and Insights

As outlined earlier, our dynamic symbolic execution performs a symbolic execution of the program simultaneously with its concrete execution. For a method under examination, all class instance and static variables, the method's parameters, and the method's result are treated as symbolic variables. The path condition of the symbolic execu-

tion is determined purely by the paths taken in the concrete execution—no exploration of other paths using symbolic values is performed. When executing a single test case, the path condition at the end of the symbolic execution represents the symbolic condition for the path the program followed. Thus, the path condition corresponds exactly to a precondition *for that particular test case execution.* Similarly, the symbolic values of the method's result and of the object instance variables form the method's postcondition for the specific test case. Repeating the process for all test cases, we get a collection of preconditions and postconditions, which all need to hold for the method. Combining the preconditions and postconditions for all test runs, we obtain the total precondition and postcondition of the method. The conditions are simplified through symbolic reasoning before being presented to the user. Individual conditions (i.e., without logical disjunction—see later) that concern only instance variables (i.e., no parameters) and that hold on entry and exit of all methods are reported as class invariants.

This general scheme elides several important elements. The first interesting point concerns how conditions are combined. Consider the following simple method from the `StackAr` benchmark, described in Section 5.

```
public Object top() {
  if(Empty)
    return null;
  return theArray[topOfStack];
}
```

Imagine that we execute this method for two test cases: first on an empty stack and then on a non-empty one. The first execution produces a path condition "`Empty == true`". (`Empty` is a C# "property", therefore taking its value results in calling a method, which checks the value of `topOfStack`. Nevertheless, this method is pure so our system uses `Empty` as a logical variable in conditions instead of expanding it, as we discuss later in Section 4.2.) The path condition becomes the precondition of the method for this test case. Similarly, the postcondition is "`\result == null`", again only for this particular execution. The test case of a non-empty stack produces a precondition "`Empty == false && topOfStack >= 0 && topOfStack < theArray.Length`". The corresponding postcondition is "`\result == theArray[topOfStack]`".

Combining preconditions is done by taking the disjunction (logical-or) of the individual test cases' preconditions. In this example, the combined precondition becomes:

```
Empty == true ||
(Empty == false && topOfStack >= 0 &&
 topOfStack < theArray.Length)
```

Similarly, postconditions are combined by taking their conjunction but appropriately predicated with the corresponding precondition. Following common convention, we report the conjunction of postconditions as two separate postconditions. In our example, the inferred postconditions become:

```
Empty == true ==> (\result == null)
```

and

```
(Empty == false && topOfStack >= 0 &&
 topOfStack < theArray.Length)
==>  (\result == theArray[topOfStack])
```

Combining invariants by disjunction, conjunction, and implication brings out an interesting feature of our approach. Consider method preconditions. The crux of every dynamic invariant inference system is its *abstraction* technique. Given a method `void m(int i)` and test input values from 1 to 1000, the most precise precondition that an invariant system can infer is by disjunction—i.e., "`i == 1 || i == 2 || ... || i == 1000`". Generally, the system can be precise by inferring one disjunct for every test case executed, and combining them to form the complete precondition. Nevertheless, this precision means that dynamic observations do not generalize to other test inputs that have not been already encountered. The value of an invariant inference tool is exactly in this generalization. Thus, traditional dynamic invariant inference tools (such as Daikon or DIDUCE) often avoid combining observations precisely using disjunction and instead try to generalize and abstract. For instance, a reasonable abstract precondition for the above inputs is "`i > 0`". Once conditions have been abstracted sufficiently, they can be combined precisely across test cases using disjunction. The tool is overall responsible for heuristically deciding when to use disjunction and when to abstract away from concrete observations. The typical result is that dynamic invariant inference tools use disjunction (i.e., multiple cases) sparingly, instead preferring to generalize, which often leads to over-generalization. Instead, our approach employs no such heuristics. Our observations are already generalized, since they correspond to branch conditions in the program text, appropriately modified in the course of symbolic execution. Thus, they can freely be combined precisely using disjunction. Even if there is a large number of test inputs, the number of disjuncts in our output is bounded by the program paths in the method under examination. (Of course, the number of program paths can be infinite in the case of loops, and we have to apply special abstraction techniques, discussed later in the paper.)

Another interesting feature of the dynamic symbolic execution technique is that some relatively "deep" invariants can be easily established. For the above example, our DySy tool easily infers the postcondition *pure*, indicating that the method has no effects visible to its clients. In contrast, traditional dynamic invariant inference tools treat the method as a black box, and can only establish shallow properties with observations at its boundaries. For instance, Daikon infers several shallow purity properties for the above example, such as "`theArray == \old(theArray)`". It cannot, however, establish the full purity of the method relative to all reachable heap data (e.g., with respect to the elements held inside the array, and all the elements referenced by them, etc.).

Finally, the dynamic symbolic execution approach to invariant inference is heavily dependent on a symbolic reasoning engine (e.g., a theorem prover) for producing output that is close to the expectations of a human user. Without symbolic simplification of conditions, invariants end up too verbose, with multiple tautologies. For a simple example, consider a method `allInts` with the following structure:

```
void allInts(int i) {
  if (i < 0)
  { ... } // do something
  else if (i == 0) { ... } // do something else
  i++;
  if (i > 1) { ... } // do something else
}
```

If the program's regression test suite exercises all paths, then it is natural to expect a precondition of `true` rather than the unreduced "`((i < 0) && !(i+1 > 1))
|| (!(i < 0) && (i == 0) && !(i+1 > 1))
|| (!(i < 0) && !(i == 0) && (i+1 > 1))`". Thus, symbolic reasoning is necessary to establish this tautology. It is worth noting that existing dynamic invariant inference tools can also benefit from symbolic reasoning in order to simplify their reported invariants. For instance, Daikon produces several extraneous invariants for the earlier `top` routine of `StackAr`: a postcondition "`topOfStack == \old(topOfStack)`" is reported, but other postconditions include both clauses "`\result == theArray[topOfStack]`" and "`\result == theArray[\old(topOfStack)]`".

We next discuss our specific implementation of dynamic symbolic execution for invariant detection in the DySy tool. DySy benefits from the mature symbolic execution and reasoning capabilities of the Pex framework.

4.2 DySy, Pex, and Symbolic Reasoning

Pex [33] is a dynamic analysis and test generation framework for .NET, developed by the Foundations of Software Engineering group at Microsoft Research. Pex monitors the execution of a program through code instrumentation. The instrumented code drives a "shadow interpreter" in parallel with the actual program execution. For every regular .NET instruction, there is a callback to Pex, which causes the "shadow interpreter" to execute the operation symbolically. The Pex interpreter is almost complete for the .NET instruction set. It is only missing the logic to perform control-flow decisions, since it is passively monitoring the actual program execution, which performs the decision actively.

Pex's main functionality is similar to the Dart tool [17]: Pex tests programs exhaustively in a feedback loop, in which an automatic constraint solver finds new test inputs that represent execution paths that Pex did not monitor yet. While we do not use this test input generation feature in DySy, we do use Pex's capability to construct and reason about symbolic program states.

4.2.1 Background: Pex Symbolic States, Terms

A symbolic program state is a predicate over logical variables together with an assignment of terms over logical variables to locations, just as a concrete program state is an assignment of values to locations. The locations of a state may be static fields, instance fields, method arguments, locals, and positions on the operand stack.

Pex's term constructors include primitive constants (integers, floats, object references), and functions over integers and floats representing particular machine instructions, e.g., addition and multiplication. Other term constructors implement common datatypes such as tuples and maps. Pex uses tuples to represent .NET value types ("structs"), and maps to represent instance fields and arrays, similar to the heap encoding of ESC/Java [13]: An instance field of an object is represented by a single map which associates object references with field values. Constraints over the .NET type system and virtual method dispatch lookups can be encoded as well. Predicates are represented by boolean-valued terms.

Pex implements various techniques to reduce the overhead of the symbolic state representation. Before building a new term, Pex always applies a set of reduction rules that compute a normal form. A simple example of a reduction rule is constant folding, e.g., $1 + 1$ is reduced to 2. All logical connectives are transformed into a BDD representation with if-then-else terms [3]. All terms are hash-consed, i.e., only one instance is ever allocated in memory for all structurally equivalent terms.

Recall the method `top` given in an earlier example. When we execute the method and `Empty == false`, then the result of the method call, `theArray[topOfStack]`, will have the following term representation.

```
select(select(int[]_Map,
              select(theArray_Map, this)),
       select(topOfStack_Map,this))
```

where `select(`m`,` i`)` represents the selection of the value stored at index i in the map m. `theArray_Map` and `topOfStack_Map` are maps indexed over object references, so `select(theArray_Map, this)` corresponds to `this.theArray` in the source language. `int[]_Map` is a map of array references to another map that contains the elements of the array, indexed over integers.

A state update, e.g.,

```
this.topOfStack = this.topOfStack+1;
```

which method `push` may perform, is represented using an update function `update(`m`,` i`,` v`)`, which represents the map m after is was updated at index i with new value v.

```
topOfStack_Map' =
  update(topOfStack_Map, this,
         add(select(topOfStack_Map, this), 1))
```

The interpreter records all conditions that cause the program to branch. In addition to the explicit conditional branches performed by the program, Pex's interpreter also models all implicit checks performed by the runtime which may induce exceptional behavior—e.g., following a reference is treated as an implicit branch based on whether the reference is null (exceptional path) or not (normal path).

Based on the already accumulated path condition, terms are further simplified. For example, if the path condition already established that $x > 0$, then $x < 0$ reduces to `false`.

Pex has a term pretty printer which can translate back reduced and simplified terms into readable C# syntax.

4.2.2 DySy Algorithm

DySy symbolically monitors the concrete execution of a given test suite. For the duration of each method call, DySy registers a separate interpreter with Pex's monitoring framework. Thus, as soon as there are nested method calls, multiple interpreters will be listening to the callbacks of the instrumented code. DySy builds a set of quadruples (*method, pathCondition, result, finalState*) as it monitors the program. Each quadruple characterizes an execution path of a method.

Step 1: Path condition and final state discovery.

When the program initiates a call to a method M (including the `Main` method of the test suite), DySy creates a new interpreter instance along with a new symbolic state instance. DySy initializes the locations of the symbolic state, including the method's arguments, with logical variables. The interpreter will evolve the symbolic state according to all subsequently executed instructions, including transitions

into and out of other method calls. When the call to M that spawned this interpreter instance returns, DySy records the quadruple (M, *pathCondition, result, finalState*), and abandons the interpreter. The result is the term that M returns.

During nested method calls, the state's locations always hold terms built over the original logical variables of M, and the result of the call is also a term over the original logical variables. When the program performs no state updates during a (nested) call, except updates to the local variables of newly created stackframes and updates to instance fields of newly created objects, DySy considers the call *pure*. DySy replaces the result of a pure call with a term representing the call—e.g., in our earlier example it replaces the explicit result `topOfStack >= 0` with the method name `Empty`.

Also, DySy abstracts all, directly or indirectly, recursive calls to M in this way, regardless of whether they are pure calls or not. This is a heuristic treatment, which results in recursive invariants. (This avoids unbounded paths through recursive methods. The other interesting case is unbounded paths through loops, which we discuss separately in Section 4.3.) For example, the factorial function

```
int fac(int i) {
  if (i<=1)
    return 1;
  else
    return i * fac(i-1);
}
```

is eventually characterized by DySy as a method with no precondition, and the postcondition

```
\result == ((i <= 1) -> 1) else -> i*fac(i-1)
```

Pex does not know the effects of code that it does not monitor. For example, calls to "native" methods are not monitored. Here, the user can choose between a conservative and an optimistic treatment.

Step 2: Class invariant derivation.

At the end of symbolic execution and before outputting method preconditions and postconditions, DySy first computes class invariants, which are used to simplify the methods' invariants. DySy defines the set of "class invariant candidates" of a class C as the set of conjuncts c of all recorded path conditions of all methods of C, where c only refers to the `this` argument but no other argument. (For future work, one could existentially quantify the other argument symbols to gain more class invariant candidates.) For each path condition and final state of a method of C, DySy then checks which candidates are implied by all path conditions in the final states of all methods of C. (In fact, in its current implementation, DySy does not perform a precise implication check using an automatic theorem prover. Instead, it simply executes the test suite again, and checks the candidates in the concrete final state of each call to a method of C.) The implied candidates are the "class invariant" of C.

Step 3: Pre- and postcondition computation.

Finally, DySy further simplifies the method's path conditions, assuming the derived class invariant. As a consequence of this simplification, some of the quadruples might collapse together.

The precondition of a method is the disjunction of its path conditions. The postcondition of a method is the conjunction of path-specific postconditions. A path-specific postcondition is an implication with a path condition on the left hand side and a conjunction of equalities, where each equality relates a location to the term assigned to that location in the final state. (E.g., recall postcondition "`Empty == true ==> (\result == null)`" in our earlier example.)

4.3 Abstraction for Loops

Handling loops is a fundamental challenge for symbolic execution in general. In our specific context, we discussed earlier how loops result in a method having an infinite number of possible paths. Since our symbolic execution is guided by a concrete execution, every path we observe has a finite length, but grows quickly and without bounds. In practice, this means that straightforward symbolic execution produces enormous path conditions that are overly specific and defeat the purpose of using program conditions as potential invariants. We next discuss the heuristics that DySy uses for abstraction in the case of loops.

Let us examine the problem with the example of a simple linear search method, for which we want to derive invariants.

```
public int linSearch(int ele, int[] arr) {
  if (arr == null)
    throw new ArgumentException();
  for (int i = 0; i < arr.Length; i++) {
    if (ele == arr[i])
      return i;
  }
  return -1;
}
```

Consider running tests for this method with a single input array `{5, 4, 3, 12, 6}` and the numbers 0 to 9 as candidate element values. Performing symbolic execution along the path of concrete execution for `ele` equal to 0 will yield a long and too-specific path condition, even after full simplification: "`arr != null && arr.Length == 5 && ele != arr[0] && ele != arr[1] && ele != arr[2] && ele != arr[3] && ele != arr[4]`". The precondition is not only unwieldy, but also fairly bad for our purpose of inferring invariants because it does not contain general observations that may also appear in preconditions derived for other test cases: Even after all tests are run, the combined preconditions and postconditions will end up having few commonalities (e.g., "`arr != null && arr.Length == 5`") and 5 separate cases. (Four cases correspond to the four numbers from 0 to 9 that appear in the array, and one case corresponds to the path for numbers that are not found in the array.) This is exactly the "precise but useless" invariant that dynamic invariant inference aims to avoid, as discussed in Section 4.1. The problem is that our technique is based on using program conditions to partition the abstract space of possibilities into a few general but interesting categories. These coarse partitions can then be combined together with disjunctions (i.e., case-analysis). When the partitions become too fine, there is no abstraction benefit and the invariants describe exactly the behavior of the test inputs and little more.

The general approach to dealing with such over-specificity in program analysis is to force abstraction by forgetting some of the information in the too-precise program paths. We can do this by collapsing conditions together (e.g., one condition per-program-point) or by turning program variables

into unknowns (i.e., symbolic values) if they get assigned more times than a given threshold. The ideal solution would be to produce a concise strongest loop invariant condition. This is generally infeasible, although the rich research results on automatic techniques for deriving loop invariants (e.g., [4, 14, 28]) are applicable to the problem. DySy currently uses a simple heuristic that does not involve an attempt for invariant inference, only local collapsing of conditions. We first recognize loop variables by treating specially the common code pattern of `for` loops that introduce explicit variables. Loop variables are then treated as symbolic values, and a loop's exit condition does not become part of the path condition if the loop body is entered at all. Furthermore, symbolic conditions inside the body of the loop are collapsed per-program-point with only the latest value remembered: If a certain `if` statement in a loop body evaluates to true in one iteration and to false in another, the latest condition replaces the earlier one in the path condition. This effectively treats a loop as if it were an `if` statement with the symbolic conditions in the loop body collapsed per-program-point.

To illustrate the approach, our `linSearch` example uses a `for` loop with loop variable `i`, declared explicitly in the `for` loop's initialization expression. This signals to DySy that variable `i` will likely be assigned multiple values, and will participate in conditions. DySy then treats `i` as a symbolic value and does not keep track of its state updates. By itself, this would be insufficient: executing the loop and then exiting would produce contradictory symbolic conditions. In our example, we would have "`i < arr.Length`" (for the part of the path executing the loop body) and "`!(i < arr.Length)`" (when the same path later exits the loop body). Since both conditions are conjoined (logical-and) together in the same path condition, the path condition becomes just `false`, which is clearly erroneous. In our heuristic, we ignore a loop's exit condition (unless the loop is not entered at all). In our example, the precondition becomes:

```
arr != null &&
($i < arr.Length && !(ele == arr[$i]) && $i >= 0 ||
 $i < arr.Length && ele == arr[$i] && $i >= 0 )
```

This demonstrates a few interesting points. First, symbolic variable `$i` is treated as a pseudo-input. Essentially, in the above logic formula, `$i` is existentially quantified: there exists some `$i` with these properties. Second, no condition "`$i >= arr.Length`" is output. Every test case enters the loop at least once. Third, we can see how path conditions are collapsed per-program-point inside the loop body: Executions that do find the searched element produce both the condition "`!(ele == arr[$i])`" (for iterations over other elements) and the complement, "`ele == arr[$i]`" (for the iteration that finally finds the element). Yet the former are replaced when the latter take place. Finally, this precondition contains redundancy. It covers the complementary cases of "`ele == arr[$i]`" and "`!(ele == arr[$i])`", which can be simplified away. It is fortunate, however, that the DySy simplifier misses this opportunity because this helps illustrate how the different cases arise. The separation of the cases does not matter for the method's precondition, but does matter for the *postcondition*. There, we obtain (slightly simplified):

```
!(ele == arr[$i]) ==> \result == -1 ||
ele == arr[$i]  ==> \result == $i
```

This is a quite informative postcondition for the method, and captures its essence accurately.

In the immediate future, we plan to refine our heuristic into a slightly more sophisticated version that handles more than `for` loops and also produces useful conditions for exiting the loop body. Specifically, we intend to recognize loop variables by observing program variables that get assigned during the loop's iterations. Each of these program variables will give rise to two symbolic variables—e.g., `$i0` and `$i1`. The first symbolic variable will represent the values of the program variable in the body of the loop, while the second will represent the value on exit from the loop. These variables are again existentially quantified: our conditions will only reflect that there is some value `$i0` (resp. `$i1`) for which the symbolic conditions derived while executing the loop body (resp. when exiting the loop) hold.

5. EVALUATION

We next discuss and evaluate DySy in comparison with the Daikon dynamic invariant inference tool.

5.1 Discussion

At a high level, our discussion of the dynamic symbolic approach should give the reader a qualitative idea of the comparative advantages of DySy. Every dynamic invariant inference process captures, to some extent, the peculiarities of the test suite used. Nevertheless, our symbolic approach has a smaller risk of being overly specific, since the conditions themselves are induced by the program text and refined through symbolic execution. Instead, an approach observing arbitrary, pre-set conditions at method boundaries is bound to be "fooled" much more easily. For the `linSearch` method of the previous section, with the test cases described earlier (all numbers 0..9 searched in the array `{5, 4, 3, 12, 6}`) Daikon infers almost no useful invariants, but a large number of spurious ones. Example "accidental" invariants include "`size(arr[]) in arr[]`", "`size(arr[])-1 in arr[]`", "`arr[i] != i`", etc. (These relate the index or size of an array to its contents!) Certainly these spurious invariants can be disqualified with a more extensive test suite that uses more arrays as inputs. Nevertheless, test suites encountered in practice tend to exercise as many different cases in the program logic as possible, but without much variety of data. It is, thus, very plausible for a programmer to unit-test method `linSearch` with only a single array, yet with multiple input search values. A larger test input (e.g., a system-test) that exercises `linSearch` may also fail to invalidate some of the false invariants—for instance, "`arr[i] != i`" is likely to hold for many arrays.

On the other hand, a possible threat for DySy compared to Daikon is that interesting conditions are not reflected in the program text. For instance, an interesting concept, such as ordering, may be implicit or hard to infer from program conditions, yet may be inferable by Daikon. Nevertheless, we have not found this to often be the case. We believe that this is not surprising: finding an implicit interesting concept by unguided search over a space of pre-set invariant templates is quite unlikely.

5.2 A Case Study

We evaluate DySy by replicating a case study analyzed in the Daikon literature. The `StackAr` class was an example program originally by Weiss [34], which is included as the

Table 1: How many of the "ideal" invariants Daikon and DySy infer for StackAr methods and constructors exercised by the test suite. (Higher is better.) "Goal inv" is the number of our manually determined ideal invariants. "Recognized inv" is the number of these ideal invariants inferred by Daikon and DySy. For each tool, we report a strict and a relaxed count (the numbers in parentheses) because of object equality invariants. If the tool does not establish the deep equality of objects (or full purity of a method), but does establish some shallow equality condition (e.g., reference equality, or value equality up to level-1) then the "relaxed" number in parentheses counts this as matching the expected invariant.

	Goal inv	Recognized inv Daikon	DySy
Invariant	5	5	4
Constructor	3	3	2
push	4	2 (4)	2 (4)
top	3	1 (3)	2 (3)
topAndPop	4	2 (4)	2 (4)
isEmpty	3	2 (3)	3
isFull	3	2 (3)	3
makeEmpty	2	2	2
Total	27	19 (27)	20 (25)

main example in the Daikon distribution. `StackAr` is a stack algebraic data type implemented using an array. Ernst et al. [10] examine `StackAr` in detail and discuss Daikon's ability to infer `StackAr`'s invariants. In order to perform a comparison with DySy, we rewrote `StackAr` in C# (also with the help of the Java Conversion Assistant in the Visual Studio IDE).

We ran Daikon on the test suites supplied for `StackAr` by the Daikon authors. To do a comparison of Daikon and DySy, we needed an "ideal" reference set of invariants for `StackAr`. Before beginning our experimentation, a human user hand-produced our reference invariants. Inspection reveals that this set of invariants is comprehensive and minimal (in informal terms). It captures the behavior of each method in terms expected by human users. (We discuss specific examples later.)

Running DySy on the test suite takes 28 seconds, compared to 9 seconds for Daikon (2.2 seconds monitoring and 6.7 seconds inference reported) on a 2 GHz AMD Athlon 64 X2 dual core 3800+ with 4 GB of RAM. Generally, our symbolic execution adds significant overhead, which, however, is strictly lower than that of concolic execution [17, 30]. This is fast enough for real use on specific program units. Generally, we believe that the matter of invariant quality is much more significant than that of runtime overheads, as there is substantial potential for optimizations in the future.

The results of the DySy and Daikon inference are summarized in Tables 1 and 2. Table 1 shows the number of ideal invariants that were actually detected by Daikon and DySy. As can be seen, the test suite is quite thorough and both tools detect the vast majority of the target invariants. An interesting issue concerns object equality (and method purity), which is often part of the ideal invariant. The meaning of equality in our human-produced invariants is deep equality. This is not always inferred by the tools, but reference equality is more often inferred (which cannot preclude that the members of an object changed). The table offers a strict and a relaxed count. The strict count considers the invariant found even if only reference equality is established.

Table 2: Metrics on all reported invariants (lower is better), compared to ideal reference set. "Goal inv" is the number of ideal invariants. "Daikon inv" is the number of invariants reported for Daikon. "Unique subexpr" are the unique subexpressions produced by Daikon and DySy to present their invariants to the user. The total expression count is relative to the entire class so here it is less than the sum. (Some subexpressions are common across methods.)

	Goal inv	Daikon inv	Unique subexpr Goal	Daikon	DySy
Invariant	5	8	26	26	16
Constructor	3	7	17	24	17
push	4	21	28	69	43
top	3	22	14	81	25
topAndPop	4	41	21	145	50
isEmpty	3	13	9	53	9
isFull	3	11	13	45	13
makeEmpty	2	15	5	47	22
Total	27	138	89	316	133

Although both tools infer the required invariants for this test suite, the benefit of DySy is demonstrated in its avoiding *irrelevant* invariants. Table 2 shows how many total invariants Daikon inferred (third column). To detect the 27 ideal invariants, Daikon produced a total of 138 invariants. We do not give a similar count for DySy, since its output consists of condensed expressions (e.g., if-like constructs join together invariants into a single top-level one) which make the comparison uneven. Instead, we list a more reliable metric for both tools' output: The last three columns of Table 2 present the number of unique subexpressions in the ideal and inferred invariants. We parse the output for both tools and count the number of unique subtrees in the abstract syntax tree: if a subtree/subexpression occurs on two branches, it is counted only once. Thus, surface verbosity is ignored: what is measured is the number of truly distinct clauses that each tool infers. (Measuring the full size of the output would bias the numbers in favor of DySy, as its output is simplified symbolically with common subexpressions factored out.) As can be seen, DySy infers many fewer total invariants than Daikon—about a third of the total size. Indeed, the DySy output is very close to the reference set of invariants for `StackAr`. (There are minor inaccuracies in our counting of unique subexpressions, due to manual conversions between the tools' differing output syntax. When in doubt we favored Daikon, by underapproximating the number of unique subexpressions Daikon reports.)

To see an example of the differences, consider method `topAndPop`, which removes and returns the stack's most recently inserted element, or null if the stack is empty. The two important postconditions for this method concern its effect on `topOfStack` and its return value. We have:

```
\result == ((Empty -> null)
            else -> theArray[\old(topOfStack)])
```

and

```
topOfStack == ((Empty -> \old(topOfStack))
               else -> \old(topOfStack) - 1)
```

Both DySy and Daikon infer these postconditions. One more precondition states that all stack contents below the top element remain unchanged by the method's execution. Both tools infer that precondition but only under shallow equality. At the same time, to infer these correct invariants, Daikon infers a total of 41 invariants for this method. Many range from erroneous to irrelevant from the perspective of a human user. One invariant is:

```
\old(this.topOfStack) >= 0)  ==>
 (this.theArray.getClass() != \result.getClass())
```

The invariant relates the type of the array with the types of elements it holds. Another Daikon invariant is:

```
\old(this.topOfStack) >= 0)  ==>
 ((\old(this.topOfStack) >>
   stackar.StackAr.DEFAULT_CAPACITY == 0))
```

This relates `topOfStack` with the stack's default capacity using a bit-shift operator! (We hand-translated the above invariants to JML. They were originally only output in Daikon's dedicated invariant language because they are not allowed in JML—e.g., because of references to private fields.)

Eliminating the extraneous Daikon invariants would be possible with a larger test suite that would exercise the `StackAr` functionality under many conditions. Nevertheless the fundamental tension remains: If Daikon is to infer all true invariants, it needs to explore a great number of invariant templates, which increases the probability of accidental invariants. In contrast, DySy obtains its candidate invariants directly from the program's conditions and assignments, therefore the invariants it infers are very likely relevant.

6. RELATED WORK

We have already discussed the most directly related work throughout the paper. We next present some less directly related work that still exerted influences on our technique, yet either uses exclusively static methods for invariant inference, or infers program specifications purely dynamically, by examining pre-defined patterns.

For reverse engineering, Gannod and Cheng [15] proposed to infer detailed specifications statically by computing the strongest postconditions. Nevertheless, pre/postconditions obtained from analyzing the implementation are usually too detailed to understand and too specific to support program evolution. Gannod and Cheng [16] addressed this deficiency by generalizing the inferred specification, for instance by deleting conjuncts, or adding disjuncts or implications. Their approach requires loop bounds and invariants, both of which must be added manually.

Flanagan and Leino [12] propose a lightweight verification-based tool, named Houdini, to statically infer ESC/Java annotations from unannotated Java programs. Based on preset property patterns, Houdini conjectures a large number of possible annotations and then uses ESC/Java to verify or refute each of them. 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.

Taghdiri [31] 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.

Henkel and Diwan [19] have built a tool to dynamically 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.

Much of the work on specification mining is targeted at inferring API protocols dynamically. Whaley et al. [35] 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. [1] use a learner to infer nondeterministic state machines from traces; similarly, Yang and Evans [37] built Terracotta, a tool to generate regular patterns of method invocations from observed runs of the program. Li and Zhou [24] 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.

7. CONCLUSIONS

The excitement that followed the original introduction of dynamic invariant detection in the Software Engineering world seems to have been followed by a degree of skepticism. Dynamic invariant inference tools require huge and thorough regression test suites, and infer properties that are occasionally interesting but often too simplistic. Additionally, having enough tests to eliminate false invariants does not preclude extraneous invariants, which are disappointing to a human user. In this paper we presented an approach that holds promise for the future of dynamic invariant inference: using symbolic execution, simultaneously with concrete test execution in order to obtain conditions for invariants. We believe that this technique represents the future of dynamic invariant inference. It combines the advantages of invariant inference through static analysis, with the immediate practicality of observing invariants by executing tests written by programmers who exercise valid scenarios. Furthermore, the technique is strictly an increment over prior approaches, as it adds an orthogonal dimension: It is certainly possible to combine dynamic symbolic execution with observation of properties from pre-defined templates, as in other dynamic invariant detectors. The symbolic simplification approach can then apply to both symbolically inferred invariants and invariants instantiated from templates. A complete evaluation of such a hybrid is part of future work. We hope that this will be just one of many avenues that the present paper will open for dynamic invariant detection.

Acknowledgments. This work was funded by the NSF under grant CCR-0735267 and by LogicBlox Inc.

8. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 4–16. ACM, Jan. 2002.

[2] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–180. ACM, July 2006.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proc. 27th ACM/IEEE Design Automation Conference (DAC)*, pages 40–45. ACM, June 1990.

[4] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, pages 420–432. Springer, July 2003.

[5] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254. ACM, July 2006.

[6] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *Proc. 28th International Conference on Software Engineering (ICSE), Emerging Results*, pages 861–864. ACM, May 2006.

[7] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, Apr. 2005.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. 21st International Conference on Software Engineering (ICSE)*, pages 213–224. IEEE, May 1999.

[9] 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, Feb. 2001.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[11] D. Evans and M. Peck. Inculcating invariants in introductory courses. In *Proc. 28th International Conference on Software Engineering (ICSE)*, pages 673–678. ACM, May 2006.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. International Symposium of Formal Methods Europe (FME)*, pages 500–517. Springer, Mar. 2001.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, June 2002.

[14] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 191–202. ACM, Jan. 2002.

[15] G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *Proc. Second Working Conference on Reverse Engineering (WCRE)*, pages 188–197. IEEE, July 1995.

[16] G. C. Gannod and B. H. C. Cheng. A specification matching based approach to reverse engineering. In *Proc. International Conference on Software Engineering*, pages 389–398. ACM, May 1999.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering (ICSE)*, pages 291–301. ACM, May 2002.

[19] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming (ECOOP)*, pages 431–456. Springer, July 2003.

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

[21] A. K. Kolawa. Method and system for generating a computer program test suite using dynamic symbolic execution of Java programs. United States Patent 5784553, July 1998.

[22] N. Kuzmina and R. Gamboa. Extending dynamic constraint detection with polymorphic analysis. In *Proc. 5th International Workshop on Dynamic Analysis (WODA)*, May 2007.

[23] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR98-06y, Department of Computer Science, Iowa State University, June 1998.

[24] Z. Li and Y. Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 13th International Symposium on Foundations of Software Engineering (FSE)*, pages 306–315. ACM, Sept. 2005.

[25] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26. ACM, June 2005.

[26] F. Logozzo. *Modular Static Analysis of Object-Oriented Languages*. PhD thesis, Ecole Polytechnique, June 2004.

[27] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. 10th International Symposium on Foundations of Software Engineering (FSE)*, pages 11–20. ACM, Nov. 2002.

[28] C. S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proc. 11th International SPIN Workshop*, pages 164–181. Springer, Apr. 2004.

[29] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proc. 12th International Symposium on the Foundations of Software Engineering (FSE)*, pages 23–32, Nov. 2004.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 13th International Symposium on Foundations of Software Engineering (FSE)*, pages 263–272. ACM, Sept. 2005.

[31] M. Taghdiri. Inferring specifications to detect errors in code. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 144–153, Sept. 2004.

[32] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *Proc. 8th International Conference on Formal Engineering Methods (ICFEM'06)*, LNCS. Springer-Verlag, 2006.

[33] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proc. Second International Conference on Tests and Proofs (TAP)*. Springer, Apr. 2008. To appear.

[34] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.

[35] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228. ACM, July 2002.

[36] T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering*, 13(3):345–371, July 2006.

[37] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 23–28. ACM, June 2004.