Automating Software Testing Using Program Analysis

Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, and Nikolai Tillmann, *Microsoft Research*

Michael Y. Levin, *Microsoft Center for Software Excellence*

Three new tools combine techniques from static program analysis, dynamic analysis, model checking, and automated constraint solving to automate test generation in varied application domains.

During the last decade, code inspection for standard programming errors has largely been automated with static code analysis. Commercial static program-analysis tools are now routinely used in many software development organizations.[1] These tools are popular because they find many software bugs, thanks to three main ingredients: they're automatic, they're scalable, and they check many properties. Intuitively, any tool that can automatically check millions of lines of code against hundreds of coding rules is bound to find on average, say, one bug every thousand lines of code.

Our long-term goal is to automate, as much as possible, an even more expensive part of the software development process, namely software testing. Testing usually accounts for about half the R&D budget of software development organizations. In particular, we want to automate test generation by leveraging recent advances in program analysis, automated constraint solving, and modern computers' increasing computational power. To replicate the success of static program analysis, we need the same three key ingredients found in those tools.

A key technical challenge is automatic code-driven test generation: given a program with a set of input parameters, automatically generate a set of input values that, upon execution, will exercise as many program statements as possible. An optimal solution is theoretically impossible because this problem is generally undecidable. In practice, however, approximate solutions suffice: a tool that could automatically generate a test suite covering, say, even half the code of a million-line C program would have tremendous value.

Such a tool doesn't exist today, but in this article, we report on some recent significant progress toward that goal. Although automating test generation using program analysis is an old idea,[2] practical tools have only started to emerge during the last few years. This recent progress was partly enabled by advances in dynamic test generation,[3] which generalizes and is more powerful than traditional static test generation.

At Microsoft, we are developing three new tools for automatic code-driven test generation. These tools all combine techniques from static program analysis (symbolic execution), dynamic analysis (testing and runtime instrumentation), model checking (systematic state-space exploration), and automated constraint solving. However, they target different application domains and include other original techniques.

0740-7459/08/$25.00 © 2008 IEEE

Static versus dynamic test generation

Work on automatic code-driven test generation can roughly be partitioned into two groups: static and dynamic.

Static test generation consists of analyzing a program *P* statically by reading the program code and using symbolic execution techniques to simulate abstract program executions to attempt to compute inputs to drive *P* along specific execution paths or branches, without ever executing the program.[2] The idea is to symbolically explore the tree of all the computations that the program exhibits with all possible value assignments to input parameters. For each control path *p* (that is, a sequence of the program's control locations), symbolic execution constructs a path constraint that characterizes the input assignments for which the program executes along *p*. A path constraint is thus a conjunction of constraints on input values. If a path constraint is satisfiable, then the corresponding control path is feasible. We can enumerate all the control paths by considering all possible branches at conditional statements. Assuming that the constraint solver used to check the satisfiability of all path constraints is sound and complete, this use of static analysis amounts to a kind of symbolic testing.

Unfortunately, this approach doesn't work whenever the program contains statements involving constraints outside the constraint solver's scope of reasoning. The following example illustrates this limitation:

```
int obscure(int x, int y) {
  if (x == hash(y)) return -1;      // error
  return 0;                         // ok
}
```

Assume the constraint solver can't "symbolically reason" about the function `hash`. This means that the constraint solver can't generate two values for inputs `x` and `y` that are guaranteed to satisfy (or violate) the constraint `x == hash(y)`. (For instance, if `hash` is a hash or cryptographic function, it has been mathematically designed to prevent such reasoning.) In this case, static test generation can't generate test inputs to drive this program's execution through either branch of its conditional statement; static test generation is helpless for a program like this. In other words, static test generation is doomed to perform poorly whenever perfect symbolic execution is impossible. Unfortunately, this is frequent in practice owing to complex program statements (pointer manipulations, arithmetic operations, and so on) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

Dynamic test generation,[4] on the other hand, consists of

- executing the program *P*, starting with some given or random inputs;
- gathering symbolic constraints on inputs at conditional statements along the execution; and
- using a constraint solver to infer variants of the previous inputs to steer the program's next execution toward an alternative program branch.

This process is repeated until a specific program statement is reached.

Directed Automated Random Testing (DART)[3] is a recent variant of dynamic test generation that blends it with model-checking techniques to systematically execute all of a program's feasible program paths, while checking each execution using runtime checking tools (such as Purify) for detecting various types of errors. In a DART directed search, each new input vector tries to force the program's execution through some new path. By repeating this process, such a directed search attempts to force the program to sweep through all its feasible execution paths, similarly to systematic testing and dynamic software model checking.[5]

In practice, a directed search typically can't explore all the feasible paths of large programs in a reasonable amount of time. However, it usually does achieve much better coverage than pure random testing and, hence, can find new program bugs. Moreover, it can alleviate imprecision in symbolic execution by using concrete values and randomization: whenever symbolic execution doesn't know how to generate a constraint for a program statement depending on some inputs, we can always simplify this constraint using those inputs' concrete values.[3]

Let's illustrate this point with our previous program. Even though it's statically impossible to generate two values for inputs `x` and `y` such that the constraint `x == hash(y)` is satisfied (or violated), it's easy to generate, for a fixed value of `y`, a value of `x` that is equal to `hash(y)` because the latter is known dynamically at runtime.

A directed search would proceed as follows. For a first program run, pick random values for inputs `x` and `y`: for example, `x` = 33, `y` = 42. Assuming the concrete value of `hash(42)` is 567, the first concrete run takes the else branch of the conditional statement (since 33 ≠ 567), and the path constraint for this first run is `x` ≠ 567 because the expression `hash(y)` isn't representable and is therefore simplified with

DART blends dynamic test generation with model checking to systematically execute a program's feasible program paths.

Fuzz testing, a black-box testing technique, is a quick and cost-effective method for uncovering security bugs.

its concrete value 567. Next, the negation of the simplified constraint x = 567 can easily be solved and lead to a new input assignment x = 567, y = 42. Next, running the program a second time with inputs x = 567, y = 42 leads to the error.

Therefore, static test generation is unable to generate test inputs to control this program's execution, but dynamic test generation can easily drive that same program's executions through all its feasible program paths. In realistic programs, imprecision in symbolic execution typically arises in many places, and dynamic test generation can recover from that imprecision. Dynamic test generation extends static test generation with additional runtime information, so it's more general and powerful. It is the basis of the three tools we present in the remainder of this article, which all implement variants and extensions of a directed search.

SAGE: White-box fuzz testing for security

Security vulnerabilities (like buffer overflows) are a class of dangerous software defects that can let an attacker cause unintended behavior in a software component by sending it particularly crafted inputs. Fixing security vulnerabilities after a product release is expensive because it might involve deploying hundreds of millions of patches and tying up the resources of many engineers. A single vulnerability can cost millions of dollars.

Fuzz testing is a black-box testing technique that has recently leapt to prominence as a quick and cost-effective method for uncovering security bugs. This approach involves randomly mutating well-formed inputs and testing the program on the resulting data.[6] Although fuzz-testing tools can be remarkably effective, their ability to discover bugs on low-probability program paths is inherently limited.

We've recently proposed a conceptually simple but different approach of *white-box fuzz testing* that extends systematic dynamic test generation.[7] We implemented this approach in SAGE (scalable, *a*utomated, *g*uided *e*xecution), a new tool using instruction-level tracing and emulation for white-box fuzzing of Windows applications.

SAGE architecture

SAGE repeatedly performs four main tasks.

1. The tester executes the test program on a given input under a runtime checker looking for various kinds of runtime exceptions, such as hangs and memory access violations.
2. The coverage collector collects instruction addresses executed during the run; instruction coverage is used as a heuristic to favor the expansion of executions with high new coverage.
3. The tracer records a complete instruction-level trace of the run using the iDNA framework.[8]
4. Lastly, the symbolic executor replays the recorded execution, collects input-related constraints, and generates new inputs using the constraint solver Disolver.[9]

The symbolic executor is implemented on top of the trace replay infrastructure TruScan,[10] which consumes trace files generated by iDNA and virtually re-executes the recorded runs. TruScan offers several features that substantially simplify symbolic execution. These include instruction decoding, providing an interface to program symbol information, monitoring various I/O system calls, keeping track of heap and stack frame allocations, and tracking the data flow through the program structures.

The constraint generation approach SAGE uses differs from previous dynamic test generation implementations in two main ways. First, instead of a source-based instrumentation, SAGE adopts a machine-code-based approach. This lets us use SAGE on any target program regardless of its source language or build process with little up-front cost. This is important in a large company that uses multiple source languages—both managed and unmanaged—and various incompatible build processes that make source-based instrumentation difficult.

Second, SAGE deviates from previous approaches by using offline trace-based, rather than online, constraint generation. Indeed, hard-to-control nondeterminism in large target programs makes debugging online constraint generation difficult. Thanks to offline tracing, constraint generation in SAGE is completely deterministic because it works with an execution trace that captures the outcome of all nondeterministic events encountered during the recorded run.

Generational path exploration

Because SAGE targets large applications where a single execution might contain hundreds of millions of instructions, symbolic execution is its slowest component. Therefore, SAGE implements a novel directed search algorithm, dubbed *generational search*, that maximizes the number of new input tests generated from each symbolic execution. Given a path constraint, all the constraints in that path are systematically negated one by one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver. (In contrast, a standard depth-

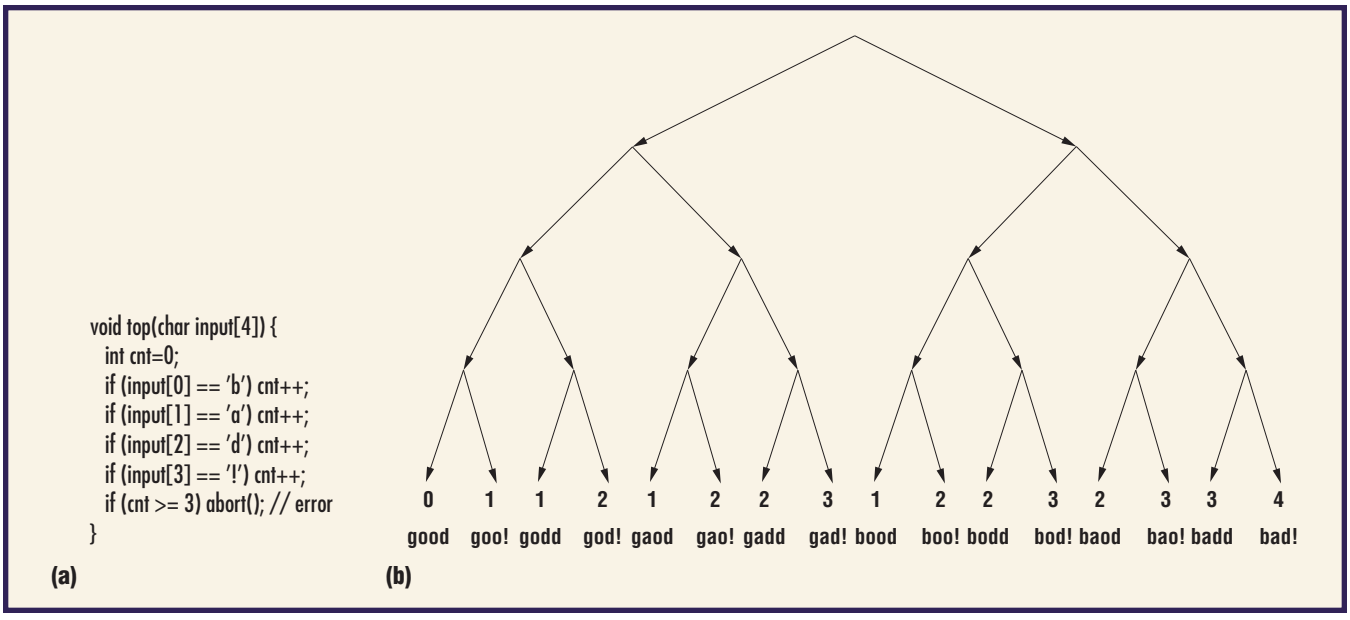


Figure 1. Example of program (a) and its search space (b) with the value of cnt at the end of each run.

or breadth-first search would negate only the last or first constraint in each path constraint.)

Consider the program in Figure 1. This program takes 4 bytes as input and contains an error when the value of the variable `cnt` is greater than or equal to three. Starting with some initial input `good`, SAGE executes its four main tasks. The path constraint for this initial run is $i_0 \neq \mathtt{b}$, $i_1 \neq \mathtt{a}$, $i_2 \neq \mathtt{d}$, and $i_3 \neq \mathtt{!}$.

Figure 1 also shows the set of all feasible program paths for the function `top`. The left-most path represents the program's initial run and is labeled with 0 for Generation 0. Four Generation 1 inputs are obtained by systematically negating and solving each constraint in the Generation 0 path constraint. By repeating this process, all paths are eventually enumerated. (See related work[7] for other benefits of a generational search as well as several optimizations that are key to handling long execution traces.)

Experience

On 3 April 2007, Microsoft released an out-of-band security patch for code that parses ANI-format animated cursors. The Microsoft SDL Policy weblog states that extensive black-box fuzz testing of this code failed to uncover the bug and that existing static-analysis tools aren't capable of finding the bug without excessive false positives.

In contrast, SAGE can generate a crash exhibiting this bug starting from a well-formed ANI input file, despite having no knowledge of the ANI format. We arbitrarily picked a seed file from a library of well-formed ANI files and analyzed a small test program that called user32.dll to parse ANI files. The initial run generated a path constraint with 341 branch constraints after parsing 1,279,939 instructions over 10,072 symbolic input bytes. SAGE then created a crashing ANI file after 7 hours 36 minutes and 7,706 test cases, using one core of a 2-GHz AMD Opteron 270 dual-core processor running 32-bit Windows Vista with 4 Gbytes of RAM.

SAGE is currently being used internally at Microsoft and has already found tens of previously unknown security-related bugs in various products.[7]

Pex: Automating unit testing for .NET

Although it's important to analyze existing programs to find and remove security vulnerabilities, automatic-analysis tools can help avoid such programming errors to begin with when developing new programs.

Unit testing is a popular way to ensure early and frequent testing while developing software. Unit tests are written to document customer requirements at the API level, reflect design decisions, protect against observable changes of implementation details, and as part of the testing process, achieve certain code coverage. A unit test, as opposed to an integration test, should only exercise a single feature in isolation. This way, unit tests don't take long to run, so developers can run them often while writing new code. Because unit tests target only individual features, it's usually easy to locate errors from failing unit tests.

Many tools, such as JUnit and NUnit, support unit testing. These tools manage a set of unit tests and provide a way to run them and inspect the results. However, they don't automate the task of creating unit tests. Writing unit tests by hand is a

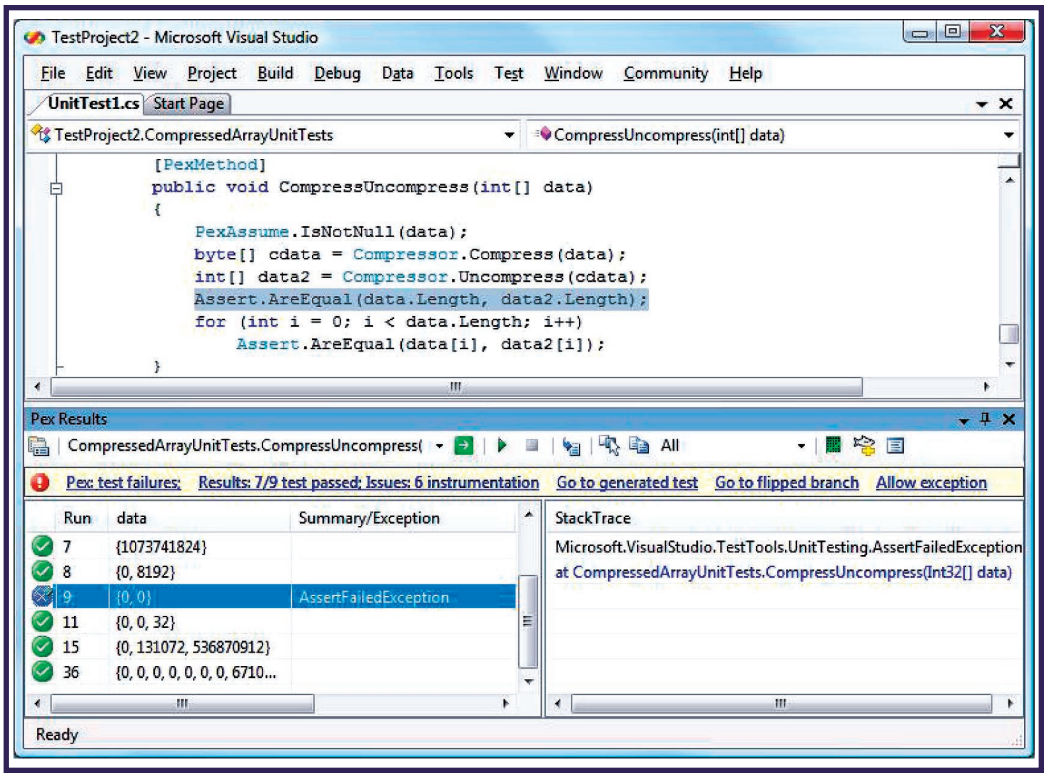


Figure 2. A glimpse of Pex in Visual Studio.

laborious undertaking. In many projects at Microsoft, there are more lines of code for the unit tests than for the implementation being tested.

On the other hand, most fully automatic test-generation tools suffer from a common problem: they don't know when a test fails (except for obvious errors, such as a division-by-zero exception). To combine the advantages of automatic test generation with unit tests' error-detecting capabilities, we've developed a new testing methodology: the *parameterized unit test* (PUT).[11]

A PUT is simply a method that takes parameters. Developers write PUTs, just like traditional unit tests, at the level of the actual software APIs in the software project's programming language. The purpose of a PUT is to express an API's intended behavior. For example, the following PUT asserts that after adding an element to a non-null list, the element is indeed contained in the list:

```
void TestAdd(ArrayList list, object element) {
  Assume.IsNotNull(list);
  list.Add(element);
  Assert.IsTrue(list.Contains(element));
}
```

This PUT states assumptions on test inputs, performs a sequence of method calls, and asserts properties that should hold in the final state. (The initial assumptions and final assertions are similar to method preconditions and postconditions in the design-by-contract paradigm.)

Pex (for *p*rogram *ex*ploration; http://research.microsoft.com/Pex) is a tool developed at Microsoft Research that helps developers write PUTs in a .NET language. For each PUT, Pex uses dynamic test-generation techniques to compute a set of input values that exercise all the statements and assertions in the analyzed program. For example, for our sample PUT, Pex generates two test cases that cover all the reachable branches:

```
void TestAdd_Generated1() {
  TestAdd(new ArrayList(0), new object());
}

void TestAdd_Generated2() {
  TestAdd(new ArrayList(1), new object());
}
```

The first test executes code (not shown here) in the array list that allocates more memory because the initial capacity 0 isn't sufficient to hold the added element. The second test initializes the array list with capacity 1, which is sufficient to add one element. Pex comes with an add-in for Visual Studio that enables developers to perform most frequent tasks with a few mouse clicks. Also, when Pex generates a test that fails, Pex performs a root cause analysis and suggests a code change to fix the bug.

Figure 2 illustrates Pex in Visual Studio. The upper code window shows a PUT; the lower window shows the results of Pex's analysis of this test combined with the code-under-test. On the lower left side is a table of generated input parameter values. The selected row indicates an assertion violation when the `data` argument is an array with two elements, each of which contain a zero. This violation is associated with a stack trace shown on the lower right side and the highlighted line in the upper code window. In SAGE, the test input is usually just a sequence of bytes (a file); in contrast, Pex generates inputs that are typed according to the .NET type system. Besides primitive types (Boolean, integer, and so on), the inputs can be arrays, arrays of arrays, value types (`struct` in C#), or instances of other classes. When the formal type of an input is an interface or an abstract class, Pex can generate mock classes with methods that return different values that the tested code distinguishes. The result is similar to the behavior of mock objects that developers would write by hand today. The following example shows a method of a mock object that can choose to return any value:

```
class MFormatProvider : IFormatProvider {
  public object GetFormat(Type type) {
    return PexOracle.ChooseResult<object>();
  }
}
```

However, the mock objects Pex automatically generates will often behave unexpectedly. When the developers aren't interested in arbitrary mock objects, they can restrict them by specifying assumptions in a style similar to assumptions on a PUT's parameters. Pex's analysis engine is based on dynamic test generation, and it uses Z3 as its constraint solver (http://research.microsoft.com/projects/Z3). After analyzing a PUT, Pex reports all the errors found. It can re-execute some of the generated tests to reproduce errors. It can also reuse the generated test suite later for regression testing.

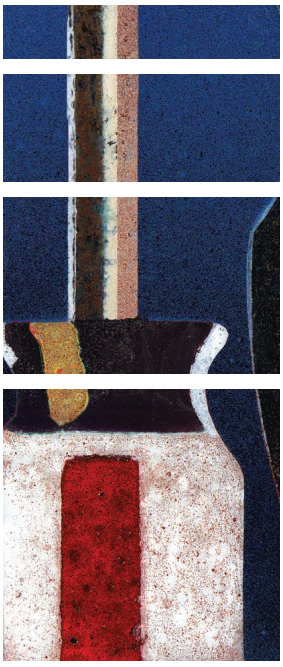
We have applied Pex to large components of the .NET framework used by thousands of developers and millions of end users. Pex found several errors, some of which were serious and previously unknown. Pex is available under an academic license on the Microsoft Research Web site (http://research.microsoft.com/Pex), and we're actively working toward a tighter integration with Visual Studio.

Yogi: Combining testing and static analysis

Testing and static analysis have complementary strengths. Because testing executes a program concretely, it precludes false alarms, but it might not achieve high coverage. On the other hand, static analysis can cover all program behaviors at the cost of potential false alarms, because the analysis ignores several details about the program's state. For example, the SLAM project has successfully applied static analysis to check the properties of Windows device drivers.[12] Thus, it's natural to try to combine testing and static analysis. For combining these, the Yogi tool implements a novel algorithm, Dash[13] (initially called Synergy), which was recently enhanced to handle pointers and procedures. The Yogi tool verifies properties specified by finite-state machines representing invalid program behaviors. For example, we might want to check that along all paths in a program, for a mutex `m`, the calls `acquire(m)` and `release(m)` are called in strict alternation. Figure 3a shows a program that follows this rule. There are an unbounded number of paths for the loop in lines 2 through 6 if the loop count `c` is an unbounded input to the program. Thus, it's problematic to exercise all the feasible paths of the program using testing.

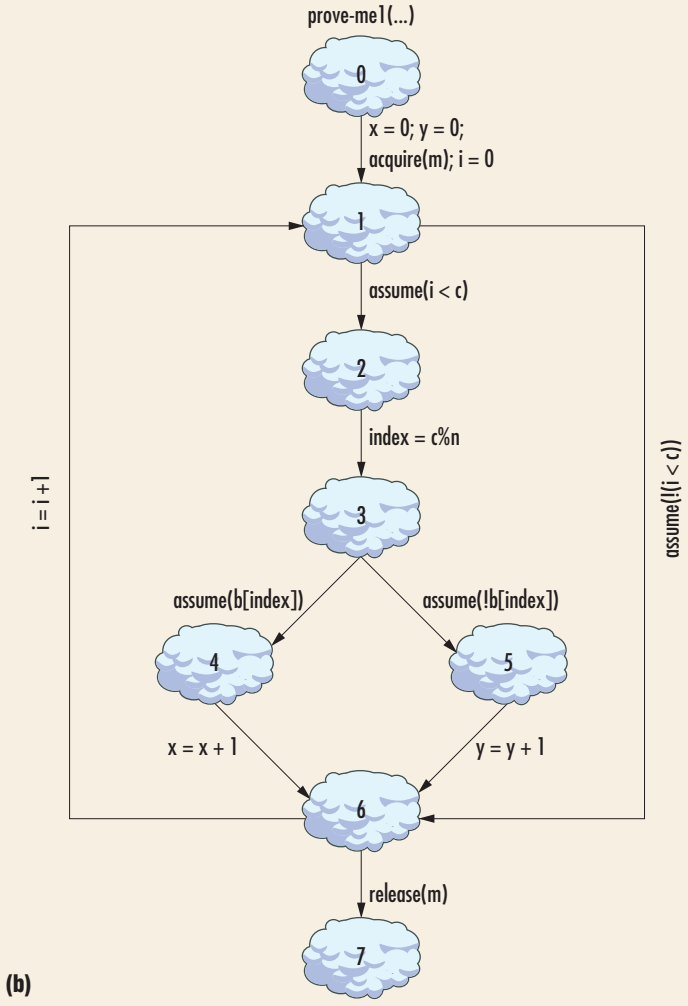
Yet Yogi can prove that `acquire(m)` and `release(m)` are correctly called along all paths by constructing a *finite abstraction* of the program that includes (that is, overapproximates) all its possible executions. A program's state is defined by a valuation of the program's variables. Programs might have an infinite number of states, denoted by Σ. The states of the finite abstraction, called regions, are equivalence classes of concrete program states from Σ. There is an abstract transition from region S to region S' if there are two concrete states $s \in S$ and $s' \in S'$ such that there is a concrete transition from s to s'. Figure 3b shows a finite-state abstraction for the program in Figure 3a. This abstraction is isomorphic to the program's control-flow graph. By exploring all the abstraction's states, Yogi establishes that the calls to `acquire(m)` and `release(m)` always occur in strict alternation.

One of Yogi's unique features is that it simultaneously searches for both a test to establish that the program violates the property and an abstraction to establish that the program satisfies the property. If the abstraction has a path that leads to the property's violation, Yogi attempts to focus test-case generation along that path. If such a test case can't be generated, Yogi uses information from the unsatisfiable constraint from the test-case generator to refine the abstraction. Thus, the construction of test cases and abstraction proceed hand in hand, using error traces in the abstraction to guide test-case generation and constraints from failed test-



```
void prove-me1(mutex *m, count c, bool b[], int n)
{
        //assume that the array b has n elements
        int x = 0; y = 0;
0:      acquire(m);
1:      for( i = 0; i < c ; i++) {
2:        int index = c%n; //assign index = c mod n
3:        if(b[index])
4:          x = x + 1;
          else
5:          y = y + 1;
        }
6:      release(m);
}
```

(a)



(b)

Figure 3. Program example (a) and a finite abstraction (b). Yogi uses static analysis and the abstraction to avoid exploring infinitely many paths.

case generation attempts to guide refinement of the abstraction. Using test cases to refine abstractions is particularly handy if the program has pointer variables that potentially *alias* each other—that is, they might point to the same object. Consider the following program:

```
struct DE {
  int lock;
  int y;
};

void prove-me2(DE *p, DE *p1, DE *p2)
{
0:  p1 = malloc(sizeof(DE)); p1->lock = 0;
1:  p2 = malloc(sizeof(DE)); p2->lock = 0;
2:  p->lock = 1;
3:  if (p1->lock == 1 || p2->lock == 1)
4:    error();
5:  p = p1;
6:  p = p2;
}
```

This program has three inputs `p`, `p1`, and `p2`, all of which are non-null pointers to `struct`s of type `DE` (with two fields `DE->lock` and `DE->y`). At lines 0 and 1, pointers `p1` and `p2` are pointed to newly allocated memory, and `p1->lock` and `p2->lock` are both set to 0. Thus, the assignment to `p->lock` at line 3 can't affect the values of `p1->lock` or `p2->lock`, and the error statement at line 4 can never be reached. Note that `p` might alias with `p1` or `p2` because of assignments at lines 5 and 6. Thus, a flow-insensitive, may-alias static analysis, as implemented in tools such as SLAM, will have to conservatively assume that at the assignment at line 2, the variable `p` may alias with `p1` or `p2`, and consider all possible alias combinations. However, Yogi can prove line 4 is unreachable while only considering the alias combination ($p \neq p1 \wedge p \neq p2$) that occurs along all concrete executions. (Because this program has only one feasible execution path, SAGE and Pex would be able to prove this, too.)

Although a simple flow-sensitive path analysis would handle this example, real C programs have lots of pointers and procedures, which make fully precise context-sensitive path-sensitive analysis problematic for large programs. Instead, Yogi leverages test executions to get precise information about pointers.

We've used Yogi to analyze several examples of Windows device drivers. Yogi was able to prove or find bugs in several device drivers where SLAM times out due to explosions in the number of aliasing possibilities. (More details on this are available elsewhere.[13])

The tools we describe here might give us a glimpse of what the future of software-defect detection could be. In a few years, mainstream bug-finding tools might be able to

generate both a concrete input exhibiting each bug found (with no false alarms) and an abstract execution trace expressed in terms of predicates on program inputs that would be useful to understand key input properties explaining the bug. Such tools would be automatic, scalable (compositional[14] and incremental), and efficient (thanks to the combination with static analysis) and would check many properties at once. They would also be integrated with other emerging techniques such as mock-object creation and software contracts, and would enable and be supported by new, more productive software development and testing processes.

References

1. J. Larus et al., "Righting Software," *IEEE Software*, vol. 21, no. 3, May/June 2004, pp. 92–100.
2. J.C. King, "Symbolic Execution and Program Testing," *J. ACM*, vol. 19, no. 7, 1976, pp. 385–394.
3. P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 213–223.
4. B. Korel, "A Dynamic Approach of Test Data Generation," *Proc. IEEE Conf. Software Maintenance* (ICSM 90), IEEE CS Press, 1990, pp. 311–317.
5. P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," *Proc. Ann. Symp. Principles of Programming Languages* (POPL 97), ACM Press, 1997, pp. 174–186.
6. J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *Proc. 4th Usenix Windows System Symp.*, Usenix Assoc., 2000, pp. 59–68.
7. P. Godefroid, M.Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," *Proc. 15th Ann. Network and Distributed System Security Symp.* (NDSS 08), Internet Society (ISOC), 2008; www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf.
8. S. Bhansali et al., "Framework for Instruction-Level Tracing and Analysis of Programs," *Proc. 2nd ACM/Usenix Int'l Conf. Virtual Execution Environments* (VEE 06), ACM Press, 2006, pp. 154–163.
9. Y. Hamadi, *Disolver: The Distributed Constraint Solver Version 2.44*, tech. report, Microsoft Research, 2006; http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf.
10. S. Narayanasamy et al., "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," *Proc. Conf. Programming Language Design and Implementation* (PLDI 07), ACM Press, 2007, pp. 22–31.
11. N. Tillmann and W. Schulte, "Parameterized Unit Tests," *Proc. 10th European Software Eng. Conf. and 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (ESEC/SIGSOFT FSE), ACM Press, 2005, pp. 241–244.
12. T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. 8th SPIN Workshop* (SPIN 01), Springer, 2001, pp. 103–122.
13. N.E. Beckman et al., "Proofs from Tests," *Proc. 2008 Int'l Symp. Software Testing and Analysis* (ISSTA 08), ACM Press, 2008, pp. 3–14.
14. P. Godefroid, "Compositional Dynamic Test Generation," *Proc. Ann. Symp. Principles of Programming Languages* (POPL 07), ACM Press, 2007, pp. 47–54.

About the Authors

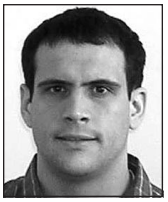
Patrice Godefroid is a principal researcher at Microsoft Research. His research interests include program specification, analysis, testing, and verification. Godefroid received his PhD in computer science from the University of Liege, Belgium. Contact him at pg@microsoft.com.

Peli de Halleux is a software design engineer at Microsoft Research. His research involves combining dynamic and static program analysis techniques for automatic test-case generation and making those accessible to the masses of developers. de Halleux received his PhD in applied mathematics from the Catholic University of Louvain. Contact him at jhalleux@microsoft.com.

Aditya V. Nori is a researcher at Microsoft Research India. His research interests include static, dynamic, and statistical analysis of programs and tools for improving software reliability and programmer productivity. Nori received his PhD in computer science from the Indian Institute of Science, Bangalore. Contact him at adityan@microsoft.com.

Sriram K. Rajamani is principal researcher and manager of the Rigorous Software Engineering group at Microsoft Research India. His research interests are in tools and methodologies for building reliable systems. Rajamani received his PhD in computer science from the University of California at Berkeley. Contact him at sriram@microsoft.com.

Wolfram Schulte is a principal researcher at Microsoft Research. His research interests include the practical application of formal techniques to improve programs' correctness and reliability. Schulte received his habilitation degree in computer science from the University of Ulm. Contact him at Schulte@microsoft.com.



Nikolai Tillmann is a senior research software design engineer at Microsoft Research. His research involves combining dynamic and static program analysis techniques for automatic test-case generation. Tillmann received his MS in computer science from the Technical University of Berlin. Contact him at nikolait@microsoft.com.

Michael Y. Levin leads the Runtime Analysis group at the Microsoft Center for Software Excellence. His interests include automated test generation, anomaly detection and debugging in distributed systems, and scalable log analysis. Levin received his PhD in computer science from the University of Pennsylvania. Contact him at mlevin@microsoft.com.