# An Empirical Study of Optimizations in YOGI

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

## ABSTRACT

Though verification tools are finding industrial use, the utility of engineering optimizations that make them scalable and usable is not widely known. Despite the fact that several optimizations are part of folklore in the communities that develop these tools, no rigorous evaluation of these optimizations has been done before. We describe and evaluate several engineering optimizations implemented in the YOGI property checking tool, including techniques to pick an initial abstraction, heuristics to pick predicates for refinement, optimizations for interprocedural analysis, and optimizations for testing. We believe that our empirical evaluation gives the verification community useful information about which optimizations they could implement in their tools, and what gains they can realistically expect from these optimizations.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Model checking*; D.2.5 [**Software Engineering**]: Testing Tools—*Symbolic execution*

## General Terms

Testing, Verification

## Keywords

Software model checking; Directed testing; Abstraction refinement

## 1. INTRODUCTION

Over the past decade we have seen several program verifiers and static analysis tools [5, 2, 1, 6] used in industrial practice. The algorithms implemented in these tools are available in published literature, and the principles behind these algorithms are well understood. However, the scalability and usability of these tools relies as much on careful engineering of optimization techniques as on the algorithms. Thus, it is important for the research community to understand which optimizations work well in practice, and how much benefit each optimization brings.

Knowledge about which optimizations work in practice is less widely known, since papers about engineering optimizations are less fashionable than papers about new algorithms. In addition, several of the tools mentioned above are commercial tools and source code for these tools is not available to the research community at large. As a result, it is very difficult for the community to replicate these verification tools. Thorough evaluation of optimizations will help a researcher building a new verification tool decide which optimizations to implement.

Part of the difficulty in evaluating optimizations is that it is very difficult to design verification tools that are bug free! For example, a bug in an optimization technique for a model checker could lead to the state space being pruned in an unsound manner. As a result, the model checker could show impressive performance gains, but the bug in the optimization could just remain unnoticed. Thus, measurements about the efficiency of optimizations can be trusted only after the verification tool itself has been tested extensively.

The goal of this paper is to empirically measure the optimizations implemented in the tool YOGI [13]. YOGI is a program verifier that checks safety properties of sequential C programs. YOGI has the capability to certify whether a program satisfies a property, and it can generate a test case to demonstrate that a program violates a property. Over the past year, YOGI has been extensively tested using benchmark suites for the Static Driver Verifier toolkit (SDV) [1]. The testing team for SDV has created several test suites over the years. One test suite consists of 619 small examples that exemplifies tool errors that have surfaced in the past. Another test suite contains 30 device drivers, and 83 properties, and hence a total of 2490 runs. The testing team has thoroughly analyzed the test suite, and determined the correct answer to each of these runs. Since YOGI has been extensively tested using these test suites, and the results inspected by several people, we have sufficient confidence about the correctness of YOGI, and believe that these test suites can be used to empirically measure the effect of optimizations. The optimizations themselves are not novel. Several of the optimizations are part of the folklore in the software model checking community, and have been implemented before in tools such as SLAM [2] and BLAST [10]. However, they have never really been thoroughly evaluated before, which is the main contribution of this paper.

The core premise behind YOGI is that static verification


*ICSE* '10, May 2-8 2010, Cape Town, South Africa

can be combined with testing to improve efficiency. The algorithms in YOGI have evolved over time. We first presented SYNERGY [9] which is the core algorithm, but it worked only for single-procedure C programs without pointers. Next, we discovered a new way to deal with pointers using the DASH algorithm [4]. More recently, we have added a way to perform interprocedural analysis such that information is reused from both tests and proofs using the SMASH algorithm [8]. Orthogonal to the algorithmic development, we have implemented a number of optimizations to make YOGI scale to large and realistic programs. In this paper, we describe these optimizations and evaluate each of these optimizations empirically. In particular, we evaluate the effect of the following optimizations:

- Initial abstraction using predicates from the property.
- Relevance heuristics to help pick suitable predicates for refinement.
- Optimizations for interprocedural analysis, including global modification analysis, and summaries for procedures.
- Using thresholding to limit the number of steps to run tests.
- Fine tuning stubs to make them executable so that YOGI can work correctly.

We present detailed empirical results for the effect of each of these optimizations over a large number of sample programs. We believe that these optimizations represent a bulk of the engineering we had to do in order to make YOGI scale and work on large programs.

## 2. OVERVIEW

A detailed description of YOGI's algorithms can be found in [9, 4, 8]. Here, we give an overview of the algorithm with just enough detail to allow us to describe the optimizations, which are the subject of this paper.

The goal of YOGI is to solve the property checking problem. An instance of the property checking problem consists of a sequential program $P$ and an assertion $\varphi$.[1] An answer to the property checking problem is "pass" if every run of $P$ satisfies the property $\varphi$. The answer is "fail" if there exists some run of $P$ that violates $\varphi$. In the latter case, the goal of YOGI is to come up with a test input $t$ such that the run of $P$ obtained by giving input $t$ violates $\varphi$. Since the property checking problem is undecidable in general, it is not possible for YOGI to give correct "pass" or "fail" answers to all instances of this problem. Specifically, YOGI might fail to terminate for certain problem instances.[2]

Static analysis and testing are complementary approaches to the property checking problem. With static analysis, we can obtain very good coverage and analyze program paths that are hard to exercise using testing, but we are forced to deal with scalability issues and false errors. With runtime testing, we can obtain only partial coverage, but the approach scales to large programs and every error that is reported is indeed realizable.

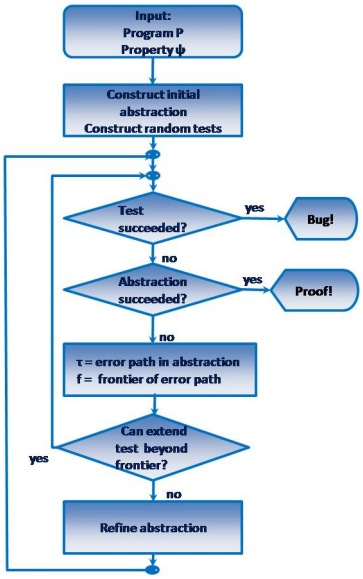


**Figure 1: The YOGI algorithm.**

YOGI's algorithm combines static analysis (based on abstractions, which are region graphs) together with testing (based on light-weight symbolic execution). YOGI simultaneously maintains a finite set of test runs and a finite abstraction of the program. A test (or test run) is a finite sequence of program states starting from some initial program state. An abstraction is a finite graph whose vertices are regions, and whose edges are over-approximations of transitions between states. Each region represents an infinite set of states. An edge exists in the abstraction from region $\sigma_1$ to region $\sigma_2$ whenever there exists states $s \in \sigma_1$ and $t \in \sigma_2$ such that there is a transition from $s$ to $t$ in the program. The set of initial states are grouped together to form an "initial" region, and the set of states that violate an assertion $\varphi$ are grouped together to form an "error" region in the abstraction. A set of tests forms an under-approximation of the set of all runs of the program, since every test is a run, and not all runs are represented in the set of tests. A region graph forms an over-approximation of the set of all runs of the program, since every run has a corresponding path in the abstraction, whereas every path in the region graph need not necessarily have a corresponding run. The goal of YOGI is to either find a test that starts from the initial region and ends in the error region, or find an abstraction that is strong enough to show that no such test exists.

A high-level view of YOGI's algorithm is shown in Figure 1. YOGI starts with an initial abstraction (which can either be the control flow graph of the program or its refinement that is based on some predicates from the property, as described below), and a set of randomly selected tests. During every iteration, if a test has managed to reach the error region, YOGI declares "test succeeded" and outputs the bug found. If no path in the abstract region graph exists from

[1] YOGI can also check safety properties expressed in the SLIC [3] language. However, these can be compiled down to assertions, so there is no loss of generality by considering assertions.

[2] In practice, we stop every run of YOGI after 1000 seconds and say that YOGI is unable to give a conclusive answer to the property checking instance.

the initial region to the error region, YOGI declares "abstraction succeeded" and outputs the current abstraction as the proof that the program satisfies the property. If neither of the above two cases are true, then YOGI finds a path $\tau$ in the abstraction from the initial region to the error region, along which a test can be potentially driven to reveal a bug. YOGI crucially relies on the notion of a frontier [9, 4], which is the boundary between tested and untested regions along the abstract counterexample $\tau$ that a test has managed to reach. In every iteration, the algorithm first attempts to extend the frontier using test case generation techniques similar to DART [7]. If test case generation fails, then the algorithm refines the abstract region graph so as to eliminate the abstract counterexample. For performing refinement, the DASH algorithm uses a new refinement operator $\mathsf{WP}_\alpha$ which is the usual weakest precondition operator that uses information from executed tests in order to compute sound and succinct refinement predicate [4].

We have now provided enough background to describe the optimizations studied in this paper.

The first set of optimizations are concerned with creating an initial abstraction that is better than the control flow graph of the program. We use predicates from the SLIC property to split regions of the control flow graph and create an initial abstraction. We can also choose other predicates to split based on dataflow dependencies on these predicates. However, the more the initial predicates, the greater is the cost (in terms of both time and space) to create the initial abstraction. This tradeoff is evaluated in Section 4.

The second set of optimizations are concerned with choosing relevant predicates for refining the abstraction. Here we study two optimizations, one that performs cheap interpolant computation, and another that uses domination relationships in the control flow graph to eliminate predicates from the program using a pre-pass (with the caveat that some of these removed predicates might be replaced later if necessary). These optimizations are described and evaluated in Section 5.

The third set of optimizations are concerned with interprocedural analysis. Techniques such as global modification analysis, as well as storing and re-using summaries from each analysis of a procedure can greatly improve the efficiency of interprocedural analysis. These optimizations are described and evaluated in Section 6.

The fourth set of optimizations are concerned with how long tests need to be run to be effective. Every transition computed using a test can potentially save a theorem prover call in the future. However, running tests also costs in terms of time and space. Section 7 evaluates the best parameter setting for how long to run tests.

The final set of optimizations are concerned with writing accurate stub functions. We painfully discovered that writing stub functions for YOGI is more challenging than writing stub functions for tools based on purely static analysis such as SLAM or BLAST. The main reason is that the stub functions for YOGI need to be precise enough to be executable for the purpose of running tests. Section 8 measures the amount of effort needed to produce executable stub functions on our benchmark test suites.

## 3. EVALUATION SETUP

Currently, the Windows driver group is evaluating YOGI as an engine to be used within Microsoft's Static Driver Verifier toolkit [1]. As a part of this evaluation, YOGI has been fully integrated as the property checking engine inside SDV (Static Driver Verifier). SLAM is another property checking engine for SDV. One of the significant advantages of this integration is that it is possible to run YOGI over large test suites that are maintained and evolved by the Windows test team. As a part of this evaluation, the Windows team has extensively compared results produced by SLAM and YOGI over this test suite. Discrepancies between the outputs of the tools have been tracked and fixed.

For the purposes of this paper, we consider two benchmark test suites from the SDV toolkit:

1. C test suite: The first test suite consists of 619 C programs that were created to test tool errors that have surfaced in SDV over the past 10 years. This suite (called SmallC suite) consists of 386 positive tests and 233 negative tests. YOGI takes approximately 20 minutes to run on this suite.

2. WDM Drivers: The second suite (called WDM suite) consists of 30 device drivers written for the Windows Driver Model (WDM) and 83 properties, a total of 2490 checks, also developed over the past 10 years. The total number of lines of code analyzed is 250 KLOC (but each driver is analyzed repeatedly 83 times for each of the properties), and YOGI takes approximately 36 hours to run on this suite on a single processor.[3]

For every run in these suites, the expected result (that is, the property should pass, or property should fail) has been manually established by the Windows group over the years. SDV itself is routinely run over several millions of lines of driver code, including drivers for the new Kernel Mode Driver Framework and User Mode Driver Framework (called KMDF and UMDF respectively), networking drivers that use the NDIS driver model, etc. The anecdotal belief in the Windows group is that any tool bugs that surface in these bigger runs are usually caught in the above two test suites [11]. Our own experience with these test suites corroborates this belief —whenever we make a mistake in implementing an optimization, these tests usually find them (it manifests as some run where the buggy optimization says that the run passes, but the property check is expected to fail on the run).

In this paper, we measure the effectiveness of an optimization by evaluating YOGI with and without the optimization on the WDM benchmark suite. We do not report performance numbers on the SmallC suite because the runs in this suite finish quickly even without any optimizations, and the optimizations have no effect on the runtime. However, the SmallC suite is very effective in catching bugs in the optimizations. We have tested each optimization we report in the paper on SmallC suite to confirm that we have not implemented the optimization erroneously.

Our earlier papers report comparisons between YOGI and SLAM [4, 13]. As a result of valuations over several hundreds of thousands of lines of code, extensive comparison of results between YOGI and SLAM, and extensive comparison between results produced by YOGI and results expected by the SDV team in these test suites, we believe that YOGI is

[3]Since each check is independent we run this on an 8-core machine usually, to get the runs to complete quickly.

```
state {
  enum {Locked = 0, Unlocked = 1} state  = Unlocked;
}

KeAcquireCancelSpinLock.Entry{
  if (state == Unlocked) {
    state = Locked;
  }
  else
     abort;
}

KeReleaseCancelSpinLock.Entry{
  if (state == Locked) {
    state = Unocked;
  }
  else
     abort;
}
```

**Figure 2: SLIC specification for CancelSpinLock property.**

currently a stable and robust tool, and we have confidence in the correctness of all the optimizations we report in this paper. Without such extensive testing, empirical evaluations of optimizations could be flawed, since buggy optimizations can boost performance due to unsound pruning of the state space of the program being analyzed.

## 3.1 Presentation methodology

All evaluations are done using 2490 runs of YOGI on the WDM suite. We make a few remarks about how we present the results of our evaluations. We group the optimizations logically so that interrelated optimizations are in the same group. Each of the subsequent sections presents a description and evaluation for one group of optimizations. For each group, we present two sets of empirical data. We present aggregate numbers for total time taken, total number of defects found, and total number of defects found for every possible choice of enabling/disabling each optimization in the group. For instance, Table 2 presents aggregate data for relevance heuristics. Since there are two relevance heuristics, we consider 4 possible choices, with or without each relevance heuristic. Thus, there are 4 rows of data in Table 2. In addition, we present scatter plots for comparing runtimes of individual runs between these 4 possible choices. As a convention, the x-axis of all scatter plots represent runtimes with *all optimizations present*. In the y-axis of each plot, we consider various choices where *at least one optimization is absent*. For example, in Section 5, we evaluate two optimizations SP and CD and show the effect of these optimization in 3 scatter plots (Figure 6, Figure 7 and Figure 8). The x-axis represents YOGI's runtimes with both the SP and CD optimizations, and the y-axis in the each of the figures represents YOGI's runtimes for the other three possible combinations of these optimizations – (without SP, with CD), (with SP, without CD) and (without SP, without CD). If most points in the scatter plots lie above the line ($x = y$), then this is an indication that the optimization is effective.

# 4. INITIAL ABSTRACTION

One possible choice for the initial abstraction used by YOGI is the control flow graph of the program. Often, we find that splitting the abstraction based on predicates present in the conditionals of the SLIC property greatly helps avoid exploring false counterexamples in the abstraction.

| Abstraction using SLIC predicates | total time (minutes) | no. defects | no. timeouts |
|---|---|---|---|
| yes | 2160 | 241 | 77 |
| no | 2580 | 241 | 86 |

**Table 1: Empirical evaluation of initial abstraction using SLIC predicates using the WDM suite.**

For example, consider the SLIC property shown in Figure 2, which specifies constraints on sequences of operations on cancel-spin-locks in the Windows kernel. At a high level, the property states that calls to `KeAcquireCancelSpinLock` and `KeReleaseCancelSpinLock` need to be performed in strict alternation. A detailed description of the syntax and semantics of SLIC can be found in [3]. For the purposes of this paper, think of a SLIC property as an object with some state (here the state is an `enum` type which can take one of two values `Locked` or `Unlocked`), and instrumentation added at appropriate points in the program to methods that manipulate the state (here instrumentation is added to the entry of calls to `KeAcquireCancelSpinLock` and `KeReleaseCancelSpinLock`).

Suppose we use YOGI to check this property on a large program. Suppose further that the program indeed satisfies the property. We find that an abstraction to establish this property needs to split all regions with the predicates `(state == Locked)` and `(state == Unlocked)`. These predicates are typically discovered by YOGI through several iterations of counterexample driven refinement. Thus, it makes sense to juts add them upfront and compute an initial abstraction in which each region in the abstraction is split with these predicates, motivating this optimization. However, there are several subtle choices in implementing this optimization. Sometimes, predicates in conditionals of SLIC properties depend on argument or return values to function calls that they are instrumenting. In such cases, it is sometimes useful to propagate the predicates to the actual parameters or the variables that gets the return value assigned at the call site. We experimented with adding such predicates as well, and we found that this resulted in actually slowing down YOGI substantially. Thus, the "sweet spot" we have arrived at is to introduce predicates based on SLIC predicates that do not depend on parameters or return values in the initial abstraction. We present empirical results to show this "sweet-spot" produces performance benefits.

## 4.1 Empirical Results

Table 1 shows aggregate metrics for running YOGI over the WDM suite with and without initial abstraction using predicates from SLIC. The first row of the table shows data from the run with the initial abstraction optimization, and the second row of the table shows data from the run without this optimization. As the results show, the time taken by YOGI improves **16%** due to this optimization, and the number of timeouts (with timeout threshold equal to 1000 seconds) reduces considerably as well.

To drill down into these aggregate results further, Figure 3 shows comparisons of runtimes of individual runs of YOGI with and without this optimization. This plot shows that while most runtimes improve due to the optimization (most points are above the straight line $x = y$), there are several

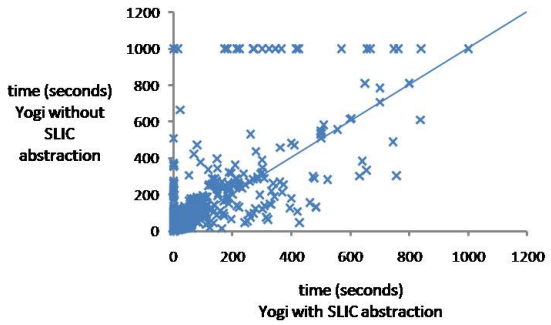


**Figure 3: Evaluating the impact of SLIC abstraction on YOGI.**

cases that degrade in runtime as well. But, the degradation never leads to a timeout, and there are several runs that timeout without the optimization that indeed complete due to the optimization. Thus, the overall effect of the optimization is very positive, and we have included it in YOGI.

# 5. RELEVANCE HEURISTICS

YOGI performs a refinement of the abstraction whenever a test cannot be extended across a frontier. Figure 4 illustrates how YOGI performs refinement. The top picture in the figure shows the frontier between regions $S_{k-1}$ and $S_k$, where a test has reached the region $S_{k-1}$ along a path through region $S_{k-2}$. The state reached by the test is denoted by "×" in the figure. Let $\beta$ denote the set of states obtained by performing forward symbolic execution along this test. Note that the set $\beta$ includes the state "×". Suppose that this test cannot be extended along the frontier. Then, YOGI performs a refinement as shown in the bottom picture of the figure. In particular, YOGI splits the region $S_{k-1}$ using a predicate $\rho$ such that $\beta \Rightarrow S_{k-1} \wedge \neg\rho$, and there is no edge in the abstraction from $S_{k-1} \wedge \neg\rho$ to $S_k$. Such a predicate $\rho$ is called a *suitable predicate*. The quality of suitable predicates inferred during verification (also related to the problem of computing invariants in program verification) is important as it can critically affect both the efficiency as well as the precision of YOGI [9, 4].

## 5.1 SP heuristic

As discussed in [4], YOGI uses the $\mathsf{WP}_\alpha$ operator to compute the suitable predicate $\rho$. $\mathsf{WP}_\alpha$ can be used to avoid considering an exponential number of aliasing conditions while performing weakest preconditions over assignments. Thus, if the operator `op` in Figure 4 on the frontier is an assignment statement, $\mathsf{WP}_\alpha$ is very helpful in producing a simple predicate $\rho$. However, if the operator `op` is of the form **assume**$(\varphi)$, we have that $\mathsf{WP}_\alpha(op, S_k) = \varphi \wedge S_k$. Thus, the $\mathsf{WP}_\alpha$ operator accumulates conditions from all assume statements along the entire path. Several of the conditionals along the path may be irrelevant to the property we are trying to prove.

In general, any interpolant [12] between $\beta$ and $\varphi \wedge S_k$ would do as a suitable predicate. Our theorem prover Z3 does not yet support computation of interpolants. We have discovered a simple relevance heuristic, called the SP heuristic, to avoid accumulating irrelevant conjuncts in the relevant predicate. The heuristic is as follows. We first check if $S_{k-1}$ is a suitable predicate. If it is, we use $S_k$. Otherwise, we use $\varphi \wedge S_k$ as a suitable predicate.

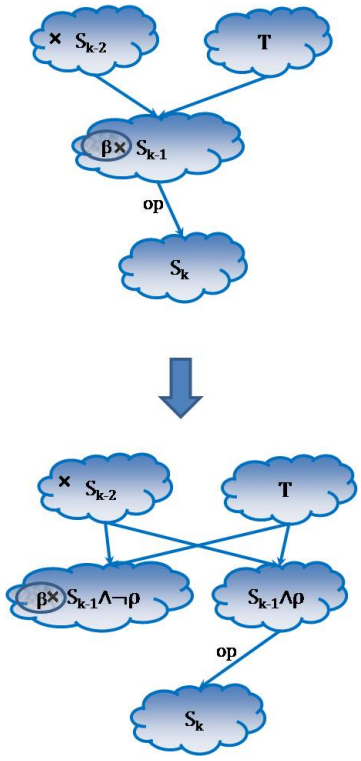


**Figure 4: Refinement using suitable predicates.**

## 5.2 CD heuristic

Though the SP heuristic is useful, it still does not fully avoid irrelevant conjuncts in suitable predicates used for refinement. We have found another orthogonal heuristic based on control-dependencies. We pre-process the input program $P$ as follows. For each **assume** statement in the program[4], we mark the **assume** statement as *potentially relevant* if some SLIC function is control dependent on the **assume** statement. Then, we make one pass over the input program $P$ and abstract every **assume** statement that is not potentially relevant using a **skip** statement, producing an abstract program $P'$. Next, we give the abstract program $P'$ to YOGI for checking. If YOGI is able to prove that $P'$ satisfies the desired property, then we have that $P$ satisfies the property as well. However, if YOGI establishes that $P'$ does not satisfy the desired property, and produces a test $t$ that executes along some control path $\gamma$ to demonstrate this, we are still not sure if $P$ violates the property. Thus, we execute the path $\gamma$ along the original program $P$ to check if it is feasible. If it is indeed feasible, we report a bug in $P$. If not, we add all **assume** statements that occur along the path $\gamma$ to $P'$ and rerun YOGI on $P'$. Note that it is possible that only a subset of the **assume** statements might suffice to explain the infeasibility of the path $\gamma$, but this subset is hard to compute in general. However, we find that adding

[4]A preprocessing step in YOGI converts all if-then-else, and looping constructs to **assume** statements.

```
1   int x;
2   void foo() {
3     bool protect = true;
4     ...
5     if(x>0)
6         protect = false;
7     ...
8     if(protect)
9       KeAcquireCancelSpinLock();
10
11    for(i = 0; i < 1000; i++){
12       // do stuff, but no calls to
13       // acquire or release CancelSpinLock
14       a[i] = readByte(i);
15    }
16
17    if(protect)
18      KeReleaseCancelSpinLock();
19  }
```

**Figure 5: Example to illustrate relevance heuristics.**

| SP heuristic | CD heuristic | total time (minutes) | no. defects | no. timeouts |
|---|---|---|---|---|
| yes | yes | 2160 | 241 | 77 |
| yes | no | 2580 | 239 | 91 |
| no | yes | 2400 | 238 | 87 |
| no | no | 2894 | 235 | 174 |

**Table 2: Empirical evaluation of relevance heuristics in YOGI using the WDM suite.**

all assume statements along an infeasible path works well in practice.

## 5.3 Example

Figure 5 shows an example to illustrate the SP and CD heuristics. Assume that the property we are interested in is the one about proper acquisition and release of CancelSpinLock from Figure 2. In this example, the call to `KeAcquireCancelSpinLock` at line 9, and the call to `KeReleaseCancelSpinlock` are guarded by conditionals which checks if the boolean `protect` is true. The other conditionals in the program, such as the check `(x > 0)` in line 5, and the check `(i < 1000)` in line 11 are irrelevant for proving the property. The SP heuristic is able to abstract the conditionals in lines 5 and 11 because there is no call to any of the SLIC functions (which, in this case are `KeAcquireCancelSpinLock` and `KeReleaseCancelSpinLock`) that are control dependent on these conditionals. The SP heuristic preserves the conditional at lines 8 and 17 since the calls at lines 9 and lines 18 to the SLIC functions are control dependent on the conditionals at line 8 and line 17 respectively.

To illustrate the SP heuristic, suppose YOGI needs to perform a refinement at line 11. Referring to Figure 4, suppose $S_k$ is the region `(state = Locked)` and `op` is the operation `assume(i > 1000)`, which is from the loop check at line 11. The weakest precondition operator $\mathsf{WP}_\alpha$ in this case will yield the conjunction of the two predicates `(state = Locked)` and `(i > 1000)`. The SP heuristic is able to eliminate the conjunct `(i > 1000)` from the refinement predicate and simplify the abstraction constructed.

## 5.4 Empirical Results

Table 2 gives aggregate metrics obtained by running YOGI over the WDM suite. As seen by comparing the first and second rows of Table 2, we see that SP heuristic improves overall

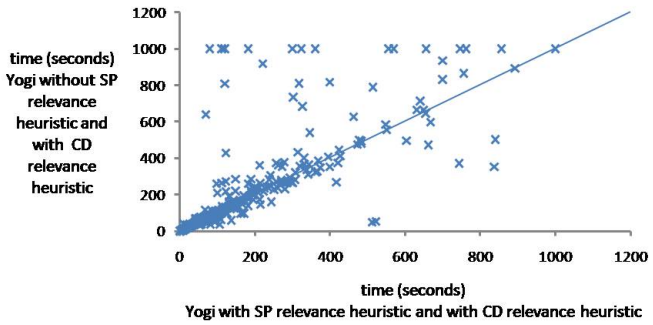


**Figure 6: Comparison of YOGI runtimes with and without the SP relevance heuristic.**

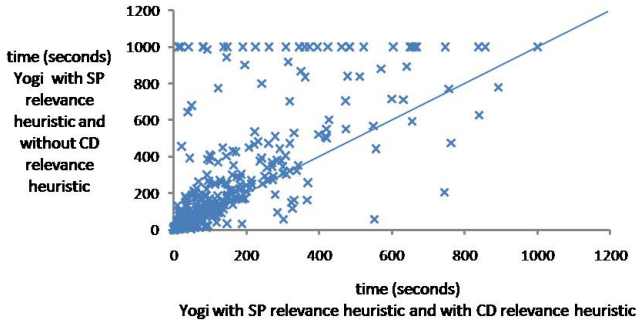


**Figure 7: Comparison of YOGI runtimes with and without the CD relevance heuristic.**

runtime of YOGI on the WDM suite by **16%**. By comparing the first and third rows, we see that CD heuristic improves overall runtime of YOGI on the WDM suite by **10%**. By comparing the first and fourth rows, we see that the combination of SP and CD heuristics improves the runtime of YOGI on the WDM suite by **25%**.

To drill down into the individual runs better, Figure 6, Figure 7 and Figure 8 compares runtimes from individual runs of YOGI with and without the SP and CD heuristics.

As seen in Figure 6 most points are close to the $(x = y)$ line, which shows that the SP heuristic does not really affect a large number of runs. This is actually due to the fact that the runtime reported for the x-axis includes the CD heuristic, which mitigates the effect of accumulation of predicates. However, there are still several runs, where the SP heuristic helps, and there are very few runs where the SP heuristic increases the runtime.

As seen in Figure 7 most points are well above the $(x = y)$ line, and there are some points below the line. We investigated the points below the line, and realized that these are because the CD heuristic sometimes aggressively abstracts the predicates. Sometimes, even if a predicate in a conditional does not directly influence the execution of a SLIC function, it could affect it indirectly by controlling assignments to some other variable, which could later influence the SLIC function. In such cases, we find that more iterations need to be spent by putting these predicates back into the program, and YOGI runs slower.

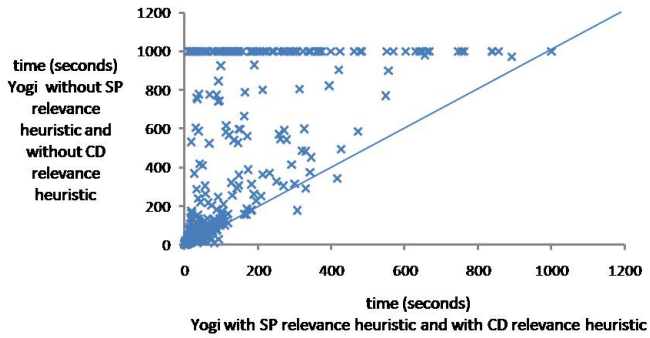


**Figure 8: Comparison of YOGI runtimes with and without both SP and CD relevance heuristics.**

However, as Figure 8 shows the combination of the two heuristics works very well, and there are only a negligible number of points below the $(x = y)$ line. As a result, both these optimizations are enabled by default in YOGI.

# 6. INTERPROCEDURAL ANALYSIS

YOGI performs modular analysis, one procedure at a time. Every query to YOGI is of the form $\langle \varphi_1, P, \varphi_2 \rangle$ which means "Is it possible to execute $P$ starting with some state from $\varphi_1$ and reach a state in $\varphi_2$?". YOGI performs optimizations to reduce the cost of interprocedural analysis, and we describe these below.

Interprocedural analysis in YOGI arises in the case when the frontier is at a procedure call. Referring back to Figure 4, suppose `op` is a call to a function, say `foo`. Then, YOGI attempts to extend the test across the frontier (which is the function call), or attempts to perform a refinement of the region $S_{k-1}$. Recall that $\beta$ is the subset of states in region $S_{k-1}$ that are obtained by performing symbolic execution along the test that reaches $S_{k-1}$. As before, we desire to pick some state from $\beta$ such that it crosses the frontier, or establish that none of the states from $\beta$ can cross the frontier and perform a refinement. YOGI approaches this problem by performing a sub-query $\langle \beta, \mathtt{foo}, S_k \rangle$. To answer this sub-query, YOGI performs the following steps:

1. First check if the query can be answered using global modification analysis. Using alias analysis, YOGI precomputes for each procedure $Q$ an over-approximation to the set of all locations that $Q$ can modify. Using this information YOGI finds the weakest precondition of $S_k$ with respect to the procedure `foo`. In particular, it analyzes every l-value in the formula representing $S_k$ and finds out which of these l-values can be modified by `foo`. YOGI then constructs an over-approximation to the call `foo` where it assumes that every l-value that can be modified can take an arbitrary (nondeterministic) value. Using this over-approximation, suppose YOGI is able to prove that from the set of states $\beta$ execution of `foo` can never reach any state in $S_k$. Then, this is a sound answer, and YOGI uses this to refine the region $S_{k-1}$. In particular, a suitable predicate is found by computing $\mathsf{WP}_\alpha$ of the region $S_k$ with respect to this over-approximation, just as in the interprocedural case. If the over-approximation fails to produce a suitable predicate for refinement, then YOGI generates a test that crosses the frontier using the above over-approximation for the call (using modification analysis). However, this test is not guaranteed to reach $S_k$ at the end of the call, since the formula used for modeling the call is an over-approximation.

2. If the first step fails, then YOGI attempts to answer the query using must summaries and not-may summaries. A must summary consists of triples of the form $\langle \phi, Q, \psi \rangle$ interpreted to mean that every state in $\phi$ can reach some state in $\psi$ by executing $Q$. A not-may summary also consists of triples of the form $\langle \phi, Q, \psi \rangle$ but is interpreted to mean that no execution of $Q$ can start in a state from $\phi$ and end in a state in $\psi$. YOGI caches must summaries and not-may-summaries from every query. Recall that the current query of interest with procedure `foo` is $\langle \beta, \mathtt{foo}, S_k \rangle$. Suppose there is a must summary $\langle \phi, \mathtt{foo}, \psi \rangle$ such that $\phi \subseteq \beta$, and $S_k \cap \psi \neq \{\}$. Then, YOGI uses this to deduce that any test in $\psi$ can extend the frontier. Dually, suppose there is a not-may summary of the form $\langle \phi, \mathtt{foo}, \psi \rangle$ such that $\beta \subseteq \phi$ and $S_k \subseteq \psi$. Then, YOGI uses this to deduce that no state in $\beta$ can reach a state in $S_k$ by executing `foo`, and that $\phi$ can be used as a suitable predicate to perform refinement of the region $S_{k-1}$.

3. Finally, if both the above steps fail, then YOGI analyzes the body of `foo` to answer the query $\langle \beta, \mathtt{foo}, S_k \rangle$. YOGI then caches the results in the form of summaries, so that these summaries can be used to answer future queries at call sites to procedure `foo`.

## 6.1 Example

Consider an example where a procedure `foo` does *not* modify a global variable $x$. Referring to Figure 4, suppose YOGI encounters a situation where the operation `op` at the frontier is a call to procedure `foo`. Suppose $S_k$ is the region $(x > 5)$, and $\beta$ is the region $(x \leq 2)$. Due to modification analysis, YOGI is able to conclude that there is no way for any execution in `foo` to start in a state from region $(x \leq 2)$ and end in a state in the region $(x > 5)$. Further, since `foo` does not modify $x$, the weakest precondition of $(x > 5)$ over the entire body of `foo` is $(x > 5)$ which is a suitable predicate to refine $S_{k-1}$.

## 6.2 Empirical Results

Table 3 shows the aggregate metrics for running YOGI over the WDM suite, for various combinations of the interprocedural optimizations. For each combination, the table shows total time taken for all the runs in the WDM suite, the number of correct defects detected by YOGI, and the total number of cases where YOGI results in a timeout. The first row of the table (Modification analysis—yes, Summaries—yes)

| Modification analysis | Summaries | total time (minutes) | no. defects | no. timeouts |
|---|---|---|---|---|
| yes | yes | 2160 | 241 | 77 |
| yes | no | 2760 | 239 | 109 |
| no | yes | 3180 | 237 | 134 |
| no | no | 3780 | 236 | 165 |

**Table 3: Empirical evaluation of interprocedural analysis optimizations with YOGI.**

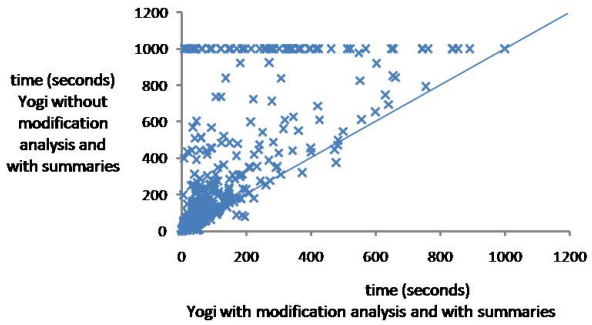


**Figure 9: Comparison of YOGI runtimes with and without modification analysis.**

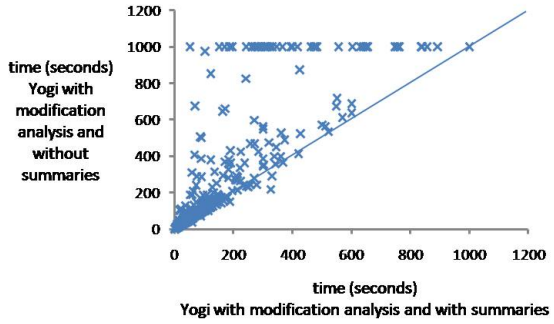


**Figure 10: Comparison of YOGI runtimes with and without summaries.**

gives empirical results with both optimizations are enabled, the second row gives results with only modification analysis was enabled (and summaries was disabled), the third row gives results with only summaries were enabled (and modification reference is not used), and the last row gives results with both optimizations disabled.

We note that both summaries and modification analysis greatly help with reducing the runtime and decreasing the number of timeouts in the runs. The total improvement due to modification analysis alone in terms of total runtime is **32%** (considering improvement from 3180 minutes to 2160 minutes), whereas the the total improvement due to summaries alone is **28%** (considering improvement from 2760 minutes to 2160 minutes). The total improvement due to both modification analysis and summaries is **42%** (considering improvement from 3780 minutes to 2160 minutes). Also, the number of runs where YOGI runs out of time or space decreases with each one of these optimizations, whereas the number of defects that YOGI finds does not change much with or without these optimizations.

To drill down into these aggregate results further, Figure 9, Figure 10 and Figure 11 show comparisons of runtimes of individual runs of YOGI with and without these two optimizations. These figures again show how useful these two optimizations are. In all the figures, the runtime of YOGI with the optimization is given in the x-axis, and the runtime of YOGI without the optimization is given in the y-axis. In all the scatter plots, almost all points above the straight

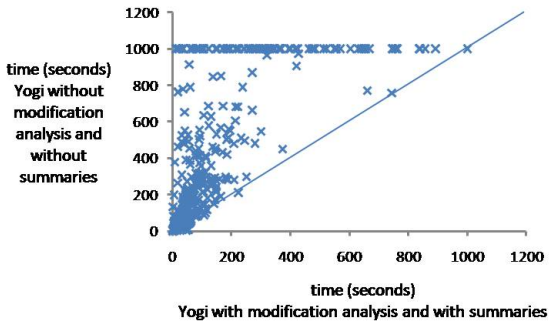


**Figure 11: Comparison of YOGI runtimes with and without both modification analysis and summaries.**

| Test threshold | total time (minutes) | no. defects | no. timeouts |
|---|---|---|---|
| 250 | 2600 | 236 | 92 |
| 500 | 2160 | 241 | 77 |
| 1000 | 2359 | 240 | 88 |
| 1500 | 2400 | 239 | 89 |

**Table 4: Empirical evaluation of test thresholds using the WDM suite.**

line $x = y$, which says that except for very few runs, there was a runtime improvement in every run due to each of these optimizations. The gains from modification analysis seem bigger than those from summaries, consistent with the aggregate numbers in Table 3. As seen in Figure 11, the combined improvement due to both the modification analysis and summaries is very significant. The row of points at the top in each of the figures represent cases where the run times out (since we have set the value of timeout to be 1000 seconds) without the optimization, whereas completes with the optimization.

## 7. TESTING

In every iteration of YOGI, testing takes precedence over verification. YOGI first checks if a test can be extended across a frontier. If this is possible, YOGI generates a test that crosses the frontier and runs the test. One parameter that needs to be tuned is the number of steps that the test should run beyond the frontier. We call this parameter as *test threshold*. The advantage in running a "long" test is that several potential future refinement queries can be avoided, since every state transition made by the test potentially avoids reasoning between the actual existence of a transition between the corresponding regions using the theorem prover. However, running the test for too long creates too many states and consumes memory for storing states. Therefore, one has to make a time vs. space tradeoff in order to choose this parameter for optimal performance of YOGI.

We have evaluated the performance of YOGI empirically for various values of test thershold and arrived at a value of **500**. Table 4 and Figure 12 compares results for test thresholds 250, 500, 1000 and 1500 respectively.

Another optimization we have implemented is to save memory while storing all the states generated from tests. For ev-

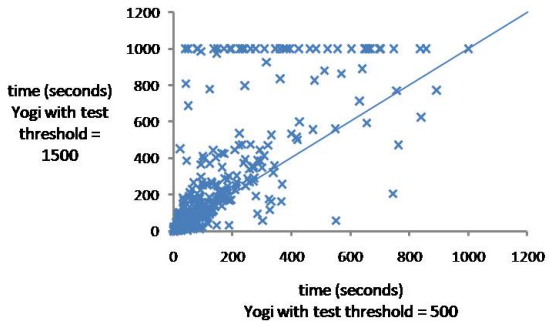


**Figure 12: Evaluating the impact of test thresholds on YOGI.**

```
1       if (DestinationString)
2       {
3           DestinationString->Buffer = SourceString;
4
5           // DestinationString->Length should be set to the
6           // length of SourceString. The line below is missing
7           // from the original stub SDV function.
8           DestinationString->Length = strlen(SourceString);
9       }
10      if (SourceString == NULL)
11      {
12          DestinationString->Length = 0;
13          DestinationString->MaximumLength = 0;
14      }
```

**Figure 13: A sample code fragment from an SDV stub function.**

| Issue type | Number of issues |
|---|---|
| Integers used as pointers | 8 |
| Uninitialized variables | 15 |
| Type inconsistencies | 9 |

**Table 5: Summary of changes made to SDV stub functions.**

ery test, YOGI generates a concrete program state at every program location in the control flow graph of the program. In order to be memory efficient, the whole concrete program state is not stored at every program location. Instead, only the *delta state*, that is, only the difference between two consecutive concrete states is stored. This poses several engineering challenges – for instance, looking up the value of a variable $x$ in a concrete state requires a series of lookup operations over a set of delta states[5]. If the value of the variable $x$ is not present in the current delta state (this can happen if $x$ is not modified at the corresponding program location), its parent is queried for the value. In the worst case, a lookup can result in traversing the test all the way up to its start state (linear time in the length of the test per lookup). Therefore, we have implemented a scheme that caches the results of lookup queries so that the amortized cost of the lookup operation is efficient (almost constant time per lookup) over all lookups.

We have found delta state optimization to be indispensable for running YOGI on large programs. Without this optimization YOGI runs out of time or space on a very large number of runs in the WDM suite. Consequently, we do not present these results. We believe it is not possible to build a tool like YOGI and make it scale without an optimization such as delta state to reduce storage at each state. An alternative is to not store states at all, and only store test inputs and rerun tests from the initial state as done in DART. However, we find that this option is also too expensive for YOGI since, unlike DART it also maintains an abstraction, and needs values from the concrete state every time a region is partitioned, so that each concrete state is assigned to the correct partition containing it.

## 8. MODELING

One of the most challenging issues we encountered in the development of YOGI is to keep the testing runs and symbolic execution runs consistent with each other. Suppose we have a test $t$ that starts at some initial state and runs along a control path $\gamma$ and results in a state $s$. Further, suppose we perform symbolic execution along the same path $\gamma$ and produce a symbolic state $\phi$. For consistency, we require that $s \in \phi$. However, such a consistency requirement is difficult to achieve in practice, in particular because the harnesses and stub functions used in our benchmarks are not type safe. Part of the reason for this is historical. Our benchmarks are obtained from Static Driver verifier and SLAM. In this context, the driver code is analyzed as is, but stub functions are written for all the OS calls that the driver makes, and a stub harness is written for how the OS loads and calls the driver. Since SLAM performs only static analysis, it was sufficient to write "rough" non-deterministic functions to model all the non-determinism in a driver's environment. For instance, if the environment returned a pointer, it was sufficient (for the purposes of doing analysis with SLAM) to write a stub that returned an integer, and later type cast it into a pointer. This had the advantage that using rough models of the environment, SLAM could still analyze the driver. However, the disadvantage was that error reports produced by SLAM could have false errors (though a lot of effort was put in to minimize this possibility). With YOGI, since bugs are established by finding test cases that lead to the error, we need higher quality OS models. Stub functions for YOGI need to be executable in the sense that running the program with the stubs should not result in memory crashes. As a result, we had to "fine-tune" all the stub functions of Static Driver Verifier so that they are executable. For example, consider the code fragment from an SDV stub function shown in Figure 13. The field `DestinationString->Length` should be set to the length of the string `SourceString` when `SourceString` is not equal to `NULL`, but is left uninitialized. This causes YOGI to miss bugs for some checks and fixing this enables it to recover these bugs.

Table 5 summarizes some of the changes that we made to the stub functions for drivers (there are a total of 456 functions for the WDM OS model). Each of these changes improved the quality of results produced by YOGI (eliminated false positives as well as false negatives). Indeed, integers used as pointers, uninitialized variables and type inconsistencies are issues due to which a program may crash thus rendering paths containing these issues infeasible.

While some of these issues (such as making sure that in-

[5]YOGI performs this operation in order to decide which abstract region a concrete state belongs to [9, 4].

tegers are not type cast to pointers, and all pointers are appropriately allocated) can be identified by doing a code-review of the stub functions, some others such as the issue in Figure 13 where we have missed assigning the `Length` field of the struct pointed by `DestinationString` under some conditions are very hard to identify by reading through the code. As a result, we discovered several of these by debugging cases with unexpected results from our test suites.

## 9. THREATS TO VALIDITY

There are two main threats to the validity of our results. Though our test suites are representative of device driver code, and we have experiential and anecdotal evidence to this effect, we do not know if these empirical results generalize to other domains. As YOGI gets used as part of SDV more inside Microsoft, we expect to be able to experiment with a wider variety of test suites, to evaluate and fine-tune these optimizations. Though our experience is that bugs in optimizations are usually caught by the test suites we have, we cannot be 100% certain about the absence of bugs in our implementation.

## 10. CONCLUSION

We have described several optimizations for YOGI and empirically evaluated them over a large number of runs from the WDM test suite. We have used empirical data to drive decisions as to what optimizations to include in YOGI. There are other optimizations that we have tried, but are absent from this paper, because they did not give good results even though they intuitively sound appealing. For instance, we had another heuristic to choose relevant predicates (see Section 5) where we favored choosing "simple" predicates over "complex" predicates, where simplicity had to do with how small the syntax tree of the predicate is, how many operators it has, etc. However, this yielded bad results, and we realized that choosing relevant predicates is more important than choosing simple predicates and this led to the SP heuristic described in Section 5.1. In another instance, we designed some sophisticated decision trees to store and retrieve not-may and must summaries, with the property that storing the summaries would be expensive and lookup would be cheap. But the performance benefit was not compelling enough.

For the optimizations that were successful, the empirical data enabled us to make decisions on fine tuning the optimization. For choosing an initial abstraction, we decided to use predicates that are present in conditionals of the SLIC specification, but do not contain references to parameters or return values. We also decided not to generate any extra predicates by propagating these predicates across assignments, by looking at empirical data. In the case of relevance heuristics, we learned that there are some cases where the CD heuristic performs worse than not having the optimization as described in Section 6. However, once we added both the CD and SP heuristics, the performance was much better. In the interprocedural case, the empirical data showed that modification analysis and summaries were not only useful individually, but their combination is very effective as well. Empirical data also enabled us to choose parameters such as test threshold.

We believe that detailed empirical study of optimizations in verification tools will enable tool builders in the community to decide on which optimizations to implement in their tools, and also have a realistic expectation on how much benefits each optimization is likely to provide.

## 11. ACKNOWLEDGEMENTS

We thank Venkatesh Prasad Ranganath and the anonymous reviewers for their insightful comments on earlier drafts of this paper.

## 12. REFERENCES

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM '04: Integrated Formal Methods*, pages 1–20, 2004.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: SPIN workshop on Model checking of Software*, pages 103–122, 2001.

[3] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08: International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP '05: European Symposium on Programming*, pages 21–30, 2005.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02:Programming Language Design and Implementation*, pages 57–68, 2002.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI '05: Programming Language Design and Implementation*, pages 213–223, 2005.

[8] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL '10: Principles of Programming Languages*, pages 43–56, 2010.

[9] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE '06: Foundations of Software Engineering*, pages 117–127, 2006.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Principles of Programming Languages*, pages 58–70, 2002.

[11] V. Levin. Personal communication, August 2009.

[12] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV '03: Computer-Aid Verification*, pages 1–13, 2003.

[13] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi Project: Software property checking via static analysis and testing. In *TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181, 2009.