Software Model Checking: Searching for Computations in the Abstract or the Concrete*

Patrice Godefroid[1] Nils Klarlund**[2]

[1] Bell Laboratories, Lucent Technologies
[2] Google

Abstract. We review and discuss the current approaches to software model checking, including the complementary views of validation versus falsification and those of static versus dynamic analysis. For falsification, also known as bug finding, we advocate the need for blended approaches that combine the strengths of both static and dynamic analysis. We outline possible directions of research in this area.

1 Introduction

Software model checking is a family of analyses that involve the automatic exploration of the state space of a program. The *state space* is at worst all the possible memory configurations that the program can read and write (such as RAM and disk space). With the combinatorial explosion that follows from this view—state spaces are often so big that "astronomical" is a powerless word to describe their sizes—the challenge is to search intelligently the state spaces of programs.

Model checking when applied to software has become a somewhat confusing concept, so we start by explaining its origin and the two complementary meanings it has come to take on. Model checking originally meant to pursue a goal complementary to testing, namely to *verify*—to assert with certainty—that the program satisfies some property. In particular, model checking in its original meaning rests on an assumption that a finite graph of manageable size representing the reachable states of the program can be constructed so that program properties can be checked by exhaustive search of this graph. Such a graph is called a *model*, let us denote it by M, and represents a set of states that are connected by transitions, each representing a program step, either in its mathematical semantics or at the machine level. A program P generates—mathematically speaking—a model M, when all possible inputs to it are considered. The term model checking comes from mathematical logic, where it means "to check whether a system is a model of a temporal logic formula". Thus, the

* This is a slightly revised version of the paper with the same title that appeared in the Proceedings of IFM'2005 (Fifth International Conference on Integrated Formal Methods), Eindhoven, November 2005, published in the Lecture Notes in Computer Science, vol. 3771, pages 20-32, Springer-Verlag.

** The work of this author was done partly at Bell Laboratories.

word "model" does not refer to a desired, abstract specification of the behavior, but to a representation of the behavior of the program. The model, in turn, describes the set of executions of the program. Each execution α is a path in the model.

2 The Main Approaches to Software Model Checking

We survey the landscape of model checking, static and dynamic analysis techniques.

2.1 The Validation View

For a property ϕ about models (programs), such as "an error state is never encountered", and an execution α of a model M, we say that α of M satisfies ϕ, and we write $\alpha, M \models \phi$, if ϕ is true of α. When we are interested in knowing whether ϕ holds for all α of M, then we say that M satisfies ϕ, or in symbols, $M \models \phi$.

In practice, it is often problematic to directly check whether $M \models \phi$ because M is anything but manageable, and can even be infinite. Instead, an approach to model checking attempts to reduce the problem to $M' \models \phi'$ involving a smaller representation M' of M and a (possibly different) formula ϕ' derived from ϕ, for example through the technique of *predicate abstraction* [18]. Whatever the algorithm is for model checking, the outcome is either "yes","don't know", or "no". If the algorithm does not provide a "yes" or "no" in reasonable time, we will treat this as a "don't know" outcome. Thus, whereas the judgment $M \models \phi$ either holds or does not hold, we will allow the judgment $M' \models \phi'$ to be undetermined.

If our emphasis is to validate a program, then it is crucial of course that an answer of "yes" implies $M \models \phi$. This criterion is *soundness* with respect to validation. For example, soundness for an abstraction algorithm that transforms the question $M \models \phi$ to the question $M' \models \phi'$ means that the implication

$$\forall M, \phi : (M' \models \phi') \Rightarrow (M \models \phi),$$

is valid.

It is also desirable but not mandatory—under this view—that if $M \models \phi$ holds then the answer delivered by the algorithm is "yes". That is, for the abstraction algorithm we would require:

$$\forall M, \phi : (M \models \phi) \Rightarrow (M' \models \phi'),$$

This criterion is *completeness* with respect to validation.

Note that a model checking algorithm that is both sound and complete never returns the answer "don't know".

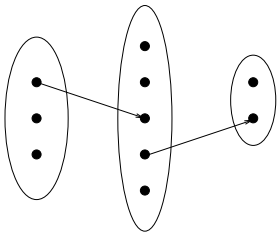

Fig. 1. May transitions

2.2 Static Analysis

Model checking under the validation view requires generating a conservative, *may-abstraction* M' of the intractable model M. Intuitively, sets of states of M are grouped together according to the abstraction to become the states of M'. For example, we may choose to disregard the value of a variable `x`, grouping together all states that are identical except for the value of `x`. Typically, the abstraction throws away enough information so that M' becomes sufficiently small. In other words, each abstract state typically corresponds to a large number of concrete states.

A may-abstraction includes all the behaviors of M and typically many more. To see this, consider the three abstract states in Figure 1. Each abstract state is a set of concrete states and are connected from the left to the right by *may* abstract transitions: each may transition stems from the existence of a concrete transition between one concrete state mapped to the origin asbtract state to another concrete state mapped to the destination abstract state. In this example, a path exists through the three abstract states, but there is no corresponding path through any corresponding concrete states.

Thus, a *may* abstraction over-approximates the set of real behaviors without running the (concrete) program, and this approach to software model checking is therefore essentially a method of *static analysis*. For instance, *type checking* can be seen as a sound, but incomplete, method of verifying that values of program variables always take their declared types. Type checking is typically incomplete since it is possible for a program to always assign at runtime values to variables according to their declared types even though the program does not pass muster with the limited reasoning powers of a type checker.

Static analysis usually focuses on soundness, not on completeness (with respect to validation). Indeed, most software model checking tools based on static analysis (e.g., [23, 7, 1, 22, 9, 6]) are incomplete: it is not always possible to guarantee that an invalidating path found in the abstracted model M' corresponds to a real execution in M.

Herein lies the weakness of predicate abstraction techniques—and other software model checking tools based on static analysis: a reported error may be illegitimate and, even if correct, it is often not accompanied by an explanation that can be readily understood by a human. In contrast, the validation engine of a type checker is usually able to pinpoint the exact place and circumstances of a

typing error; from this information, the programmer may take corrective action, such as inserting a type coercion. The best explanation in the context of static software model checking may be a program execution path fragment that may or may not be feasible under the concrete semantics. Infeasibility is often a result of the limited knowledge that a typical abstract interpretation framework has about pointers. The existence of such spurious error reports imposes an extra workload on the programmer that may lead to frustration and disuse of the tool.

Spurious error reports, often referred to as *false alarms*, can be reduced if the tool itself makes assumptions that seem reasonable. For example, a tool based on local analysis will report many uses of null pointers for input parameters of functions unless the programmer has been careful to always test input parameters locally before they are dereferenced. To avoid inundating the programmer with error messages, the tool may choose to not report errors that stem from the dereferencing of input parameters. In this way, the tool is no longer sound–from the validation point a view–but is still useful for finding bugs. Most static-analysis tools sacrifice soundness to reduce the number of false alarms, and static software model checkers are no exception.

2.3 The Falsification View

Somewhat surreptitiously, model checking in software has evolved to also denote the complementary goal to validation, namely that of *falsification*, where success is exhibiting a path that does not satisfy the property. In this sense, falsification shares goals with program testing. This change of emphasis is precipitated by the difficulty of verifying a universally quantified statement, as in the original model checking problem "does $M \models \phi$ hold?". It is sometimes easier to find an invalidating computation, that is, to solve the negation of $M \models \phi$. So, we may use "bug finding" as a synonym for "falsification".

The meaning of soundness and completeness for a program analysis algorithm become interchanged under the bug-finding view. *Soundness* with respect to falsification means that reported invalidating paths are indeed real bug traces in the sense that some set of input values does drive the program according to the path; we call the resulting bugs *sound*. *Completeness* now means that if the program is invalid then an invalidating path is indeed reported, that is, all bugs are found.

With model checking of software, we claim that most researchers have adopted (consciously or not) the falsification point of view [16]: the main practical goal of software model checking is to find bugs that would be hard to find using other techniques, and not to prove the absence of errors. So perhaps the effort would be better known as *model testing*.

2.4 Static Analysis for Falsification

One of the earliest proposals for using static analysis as a kind of program testing method was proposed by King almost 30 years ago [25]. The idea is to symbolically explore the tree of all computations a program unit (such as a

function) exhibits when all possible value assignments to input parameters are considered. For each *control path* ρ, that is, a sequence of control locations of the program, a *path constraint* ϕ_ρ is constructed that characterizes the input assignments for which the program executes along ρ. For (small) programs, all the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths ρ for which ϕ_ρ is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_ρ exactly characterize the inputs that drive the program through ρ.

A prototype of this system allowed the programmer to be presented with feasible paths and to experiment with assertions in order to force new and perhaps unexpected paths. Assuming that the theorem prover used to check the satisfiability of all formulas ϕ_ρ is sound and complete, this use of static analysis amounts to a kind of symbolic testing.

King noticed that assumptions, now called preconditions, also formulated in the logic could be joined to the analysis forming, at least in principle, an automated theorem prover for Floyd's verification method, including inductive invariants for programs that contain loops. Such a general tool for program verification has proven elusive, even after 30 years.

However, this work was followed by a rich literature on test-vector generation using static analysis and symbolic execution (e.g., see [30, 11]) and has received a renewed interest recently (e.g., [4, 3, 40, 41, 8]).

2.5 Dynamic Analysis for Falsification

Dynamic analysis operates by executing a program and observing its executions. Testing and profiling are standard dynamic analyses.

In the rest of the paper, we discuss a class of software model checkers where repeated executions, perhaps thousands or millions, are *directed* using runtime information collected dynamically through program instrumentation techniques. The program being analyzed may be a single functional unit, whose input values or formal parameters are open, or may be multithreaded and governed by a nondeterministic scheduler. In any case, the program can execute only if choices are made along the way by supplying input values and decisions about which thread to execute next. In this way, the instrumented program runs by itself, systematically or randomly making choices but—crucially—with built-in awareness that many choices lead to equivalent behaviors and only one such choice from each equivalence class needs to be considered. We mention two orthogonal equivalence concepts that have been investigated:

- *symbolic execution* that characterizes executions paths: equivalent input vectors produce computations that take the same program path (e.g., [26, 17, 5]; and
- *partial order reduction* that characterizes interleavings of a concurrent software system: equivalent interleavings produce the same significant state changes (e.g., [15, 13]).

We might classify such analyses as *directed execution*, because the analysis is execution-based but directed by analysis. Obviously, tools for directed execution (e.g., [15, 39, 29, 10, 17, 33]) are complementary to and should be used in conjunction with tools for detecting runtime errors, such as Purify [21] among many others (e.g., [31, 27, 38]).

Most software model checking methods based on dynamic analysis are sound with respect to bug finding, simply by virtue of anchoring the analysis in concrete executions of the program itself. Since an execution is a real execution, not an abstraction of an execution as in static analysis, errors encountered will usually be interesting. These analyses do not produce spurious error reports, for example suggesting impossible execution paths that are really due to a lack of computational reasoning power on behalf of the analyzer.

However, with these dynamic analyses, there is no conservative approximation or warranty typical of static analysis. In fact, the usual goal of static analysis, to consider all computations, will almost certainly not be attained. Programs are not verified, they are only tested.

We refer the reader to [12] for a general discussion on the duality and synergies between traditional static and dynamic analysis.

3 Recent Work on Directed Execution

Recently [17], we have proposed a new approach to directed execution for falsification that addresses the main limitation hampering unit testing, namely the need to write test driver and harness code to simulate the external environment of a software application. This approach combines three main techniques:

- *automated* extraction of the interface of a program with its external environment using static source-code parsing;
- automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program can operate in; and
- dynamic analysis of how the program behaves under random testing with automatic generation of new test inputs that *direct* the execution along alternative program paths.

Together, these three techniques constitute *Directed Automated Random Testing*, or *DART* for short. Thus, the main strength of DART is that testing can be performed *completely automatically* on any program that compiles – there is no need to write any test driver or harness code. During testing, DART can detect standard errors such as program crashes, assertion violations, and nontermination.

DART's integration of random testing and dynamic test generation using symbolic reasoning is best explained with an example, taken from [17].

Consider the function `h` shown in Figure 2. The function `h` is defective because it may lead to an abort statement for some value of its input vector, which consists of the input parameters `x` and `y`. Running the program with random

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
     if (f(x) == x + 10)
        abort();        /* error */
  return 0;
}
```

Fig. 2. Example of program

values of `x` and `y` is unlikely to discover the bug. The problem is typical of random testing: it is difficult to generate input values that will drive the program through *all* its different execution paths, which implies that random testing usually provides low code coverage (e.g., [32]).

In contrast, DART is able to dynamically gather knowledge about the execution of the program in what we call a *directed search*. Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths.

For the example above, the DART-instrumented `h` initially guesses the value 269167349 for `x` and 889801541 for `y`. As a result, `h` executes the then-branch of the first if-statement, but fails to execute the then-branch of the second if-statement; thus, no error is encountered. The execution defines a path ρ through the program. Intertwined with the normal execution, the predicates $x_0 \neq y_0$ and $2 \cdot x_0 \neq x_0 + 10$ are formed on-the-fly according to how the conditionals evaluate; x_0 and y_0 are *symbolic variables* that represent the values of the memory locations of variables x and y. Note the expression $2 \cdot x_0$, representing `f(x)`: it is defined through an interprocedural, dynamic tracing of symbolic expressions.

The path constraint $\phi_\rho = \langle x_0 \neq y_0, 2 \cdot x_0 \neq x_0 + 10 \rangle$ represents an equivalence class of input vectors, namely all the input vectors that drive the program through the path that was just executed. To force the program through a different equivalence class, the DART-instrumented `h` calculates a solution to the path constraint $\langle x_0 \neq y_0, 2 \cdot x_0 = x_0 + 10 \rangle$ obtained by negating, say, the last predicate of the current path constraint. A solution to this path constraint is $(x_0 = 10, y_0 = 889801541)$ and it is recorded. When the instrumented `h` runs again, it reads the values of the symbolic variables that have been previously recorded. In this case, the second execution then reveals the error by driving the program into the `abort()` statement as expected.

DART is thus a general framework parameterized by the kinds of constraints that can be collected and by their solvers. We refer the reader to [17] for a detailed presentation of the DART approach, including its formalization, several examples, the description of a simple DART implementation for the C programming language, and preliminary results of experiments. For instance, DART was

able to find automatically attacks in various C implementations of a well-known flawed security protocol (Needham-Schroeder's), as well as hundreds of ways to crash the about 600 externally visible functions provided in the oSIP library, an open-source implementation of the SIP protocol.

The directed search outlined above is closely related to prior work on dynamic test generation (e.g., [26, 20]). The main difference is that the DART approach attempts to cover *all* executable program paths in a style similar to model checking, while prior work on dynamic test generation was mostly focused on generating test inputs to exercise a *specific* program path or branch using branch/predicate classification techniques. DART is also related to test-vector generation using static analysis and symbolic execution (e.g., see [25, 30, 11, 19, 4, 3, 40, 41, 8]). Symbolic execution is limited in practice by the imprecision of static analysis and of theorem provers. As discussed in [17], DART is able to alleviate some of the limitations of symbolic execution by exploiting dynamic information obtained from a concrete execution matching the symbolic constraints, by using dynamic test generation, and by using randomization when automated reasoning is impossible or difficult. Thus symbolic execution degrades gracefully in the sense that randomization takes over, by suggesting concrete values, when automated reasoning fails to suggest how to proceed.

Independently, Cadar and Engler [5] have recently proposed a testing technique very similar to the directed search used in DART. They also describe encouraging experimental results with their implementation. CUTE [35] is a DART implementation that extends the one described in [17] by handling simple types of constraints on pointers (namely equalities and inequalities).

4 Future Work

We believe the next few years will see much research combining the static and dynamic approaches to software model checking. In this section, we outline several possible directions for future work in this area. We start this discussion with some short-term extensions to prior work on directed execution and on DART in particular. We then discuss several more open-ended problems and present specific ideas to tackle these.

4.1 Short Term

Faster constraint solvers The efficiency of DART implementations critically depends on the availability of efficient constraint solvers. This point is illustrated by the following experiment. Figure 1 compares the efficiency of two DART implementations on the Needham-Schroeder protocol benchmark (with a Dolev-Yao intruder model) discussed in [17].[1] The first implementation uses a simple

[1] All experiments were performed on a Pentium III 800Mhz processor running Linux; runtime is user+system time as reported by the Unix `time` command and is always roughly equal to elapsed time. The depth parameter limits the number of messages received by protocol entities.

depth	error?	Implementation 1	Implementation 2
1	no	5 runs (<1 second)	4 runs (<1 second)
2	no	85 runs (<1 second)	30 runs (<1 second)
3	no	6,260 runs (22 seconds)	554 runs (<1 second)
4	yes	328,459 runs (18 minutes)	9,926 runs (57 seconds)

Table 1. Impact of constraint solver on DART efficiency

constraint solver that supports only conjunctions and handles disjunctions, such as those arising from inequalities (example: $x \neq 1$ corresponds to $x < 1 \vee x > 1$), by considering in isolation each possible way of satisfying them. The second implementation uses a smarter constraint solver allowing the disjunctions that result from inequalities to be handled directly. The different approaches to inequalities reflect a seemingly innocuous choice: are two different computations that execute the same sequence of program statements to be considered distinct, in different equivalence classes, if they satisfy some disjunction in different ways? Table 1 shows the difference between a "yes" (Implementation 1) and a "no" (Implementation 2) answer to this question: for each implementation we have stated the number of executions needed to explore all equivalence classes (in order to reach an error). Table 1 shows that directly dealing with disjuncts dramatically reduces the search space. We anticipate that even faster results could be obtained for this benchmark by using general constraint solvers for handling directly all disjunctions appearing in the program's conditional statements.

Out of curiosity, we also ran (on the same machine) the static software model checker BLAST [22] (version 2.0) on this benchmark. BLAST reports "no error found" after 20 seconds for depth=1, and after 51 seconds for depth=2, but stops after reporting a spurious error after 6 minutes of search for depth=3. This spurious error is likely due to the presence of pointers in this benchmark and the limitations of the current alias analysis used in BLAST.

More constraint types and decision procedures DART reduces to random testing when no symbolic constraints on inputs can be generated. But code coverage is usually low with random testing alone, so we know that the more kinds of constraint are supported, the more effective the search for errors will be. This is true at least so long as the constraint solving itself does not become the bottleneck.

The DART framework and tool architecture are not dependent on specific constraint solvers. For instance, the first DART implementation described in [17] supported essentially integer linear constraints only, and the corresponding constraint solver used was lp_solve [28], which can solve efficiently any linear constraint using real and integer programming techniques. But we also expect to see directed execution tools that use symbolic constraints on other popular data types such as pointers, arrays, strings, or bit-vectors, in combination with more sophisticated constraint solvers, such as CVC Lite [2], ICS [24], or Simplify [36], among others. Borrowing again from static program analysis, we may use theories

for uninterpreted functions and algebraic data types to reason about frequently-used functions in specific application domains such as cryptographic libraries in security protocols.

Concurrency We have already pointed out that directed execution encompasses separate techniques for dealing with the nondeterminism of concurrency (whom to schedule) and for the mostly orthogonal issue of nondeterminism of input data (what values to provide). Indeed, concurrent programs can be sequentialized using an interleaving semantics (e.g., [15, 34]). Therefore, DART can easily be extended to multi-threaded programs and take advantage of partial-order reductions (e.g., [14, 13]). This is conceptually easy since all the threads share the same memory address space, and the formalization of [17] can be used as is. The case of multi-process programs is more complicated since a good solution requires tracking symbolic variables across processes boundaries and through operating systems objects such as message queues.

4.2 Longer Term

Combining Static and Dynamic Software Model Checking Directed execution, where the search attempts to systematically sweep all possible execution paths, may be infeasible due to combinatorial explosion. This is a particular problem if the property of interest is a localized one, such as a specific assertion in a program.

In that case, a static analysis can be used to restrict the search space to program paths that may lead to the assertion, hence eliminating irrelevant paths. This reduction can be performed using static program slicing (e.g., [37]), possibly combined with dynamic slicing, in order to prove a priori that some inputs are irrelevant.

Conversely, the precision (and hence practicality) of current static software model checker is seriously limited by the presence of calls to unknown library functions or code fragments that are beyond the capability of current symbolic execution technology such as hash functions or cryptographic functions. Because calls to libraries are frequent in systems code, this is a serious practical limitation for these model checkers, which require models for external libraries. The limitation can be alleviated by using directed execution to test the feasibility of program paths involving calls to libraries or any code that symbolic execution cannot handle easily (such as pointer-intensive code, loops, etc.). In this way, directed execution can be used by static analysis tools as a subroutine to test the feasibility of specific program paths.

Specifying Preconditions In order to effectively analyze *open* programs, an analysis tool must rely on realistic *environment assumptions*, which represent constraints on program inputs that are believed to hold.

Such constraints are also known as *preconditions*. Two broad approaches exist for specifying preconditions. The first approach consists in program annotations, usually specified directly in some fragment of mathematical logic that

is understood by the analysis tool. The second approach consists of adding code in the program itself to filter out unrealistic inputs, for instance using assertions that can be turned on for testing purposes or optionally for runtime monitoring. Ideally, one would like to combine the best of both approaches:

- specify preconditions in the host programming language (say C or Java), which is already familiar to the programmer and includes constructs for expressing sophisticated constraints on input data structures,
- yet have those preconditions interpreted without any loss of precision by the analysis tool as if they had been specified directly in logic.

We call *constraint inference* the latter problem of interpreting code as precisely as if specified directly in logic.

Note that many applications already contain input filtering code: for instance, a protocol implementation will typically first analyze the format of any incoming packet and discard it if it is not well formed. Analysis of such applications thus subsumes analysis of input filtering code; in other words, we need constraint inference capabilities.

Similarly, postconditions could be exploited, not only to check correctness of outputs, but also to direct executions towards potential postcondition violations. Moreover, the postcondition of a component is often the precondition of another one, so specifying pre- and postconditions are closely related problems.

Scalability As in traditional model checking, *state explosion* is a significant practical limitation of software model checking. Systematically exercising all executable program paths can be prohibitively expensive when the number of such paths is large (or infinite). This problem can be mitigated by *compositional testing* for large programs. For instance, consider a program P that consists of a main function f that calls exactly once another function g. If the set of inputs to f is disjoint from the set of inputs to g, the number $paths(P)$ of execution paths in P is $paths(f) * paths(g)$; but since the inputs to f are independent of those of g, both functions could instead be tested in isolation for a cost of $paths(f) + paths(g)$ while providing the same code and state-space coverage. When the inputs of f and g are not independent, compositional testing amounts of summarizing the results of g when analyzing f. The analysis of g could be summarized using pre- and postconditions constraints, in a manner similar to what is currently done in interprocedural static analysis. This form of *component summarization* extended with temporal behaviors (i.e., information about sequences of inputs/outputs at the component interface) is also similar to *assume/guarantee reasoning* in verification.

5 Conclusions

Over the last ten years, we have seen the birth of the first *software model checkers* for programming languages such as C, C++ and Java. Roughly speaking, two

broad approaches have emerged so far. The first approach consists of automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, applying traditional model checking to analyze this abstract model, and then mapping abstract counter-examples back to the code or refining the abstraction (e.g., [1, 22, 7]). The second approach consists of systematically exploring the state space of a software system by driving its executions at run-time via a scheduler and specific inputs (e.g., [15, 39, 29, 10, 17]). As discussed earlier in this paper, both of these approaches to software model checking have their advantages and limitations.

We strongly believe that, over the next ten years, an important topic of research will be combining the static and dynamic approaches to software model checking for falsification purposes. On one hand, there is a real need to improve the effectiveness of current bug finding tools, which are almost all based on imprecise, "may" static analysis and therefore prone to report (too) many false alarms, which in turn has hindered a wider adoption of these tools. On the other hand, there is a large number of static analysis techniques that have not yet been adopted to direct execution.

In this paper, we outlined several possible directions for future work in this area. We also presented specific ideas to tackle some of the key problems faced in this endeavor.

Acknowledgements

We thank Dennis Dams, Cormac Flanagan, Alan Jeffrey, Rupak Majumdar, Kedar Namjoshi, Koushik Sen, Howard Trickey, and Vic Zandy for stimulating discussions on this work. Glenn Bruns and Dennis Dams provided helpful comments on a draft of this paper. This work was funded in part by NSF CCR-0341658.

References

1. T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.
2. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of CAV'2004 (16th Conference on Computer Aided Verification)*, Boston, July 2004.
3. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of ISSTA'2002 (International Symposium on Software Testing and Analysis)*, pages 123–133, 2002.
5. C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.

6. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs . In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* , volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.
8. C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.
9. D. Dams and K. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. Technical report, Lucent Bell Labs, 2003.
10. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. *To appear in Formal Methods in System Design*, 2004.
11. J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, Linkoping, October 1999.
12. M. D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of WODA'2003 (ICSE Workshop on Dynamic Analysis)*, Portland, May 2003.
13. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of POPL'2005 (32nd ACM Symposium on Principles of Programming Languages)*, pages 110–121, Long beach, January 2005.
14. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.
15. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.
16. P. Godefroid. The Soundness of Bugs is What Matters (Position Paper). In *Proceedings of BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, Chicago, June 2005.
17. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.
18. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, June 1997. Springer-Verlag.
19. E. Gunter and D. Peled. Path Exploration Tool. In *Proceedings of TACAS'1999 (5th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1579 of *Lecture Notes in Computer Science*, Amsterdam, March 1999. Springer.
20. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.
21. R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

22. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, January 2002.
23. G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of the 21st International Conference on Software Engineering*, pages 597–607, 1999.
24. Ics. web page: `http://www.icansolve.com/`.
25. J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.
26. B. Korel. A dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.
27. E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.
28. lp_solve. web page: `http://groups.yahoo.com/group/lp_solve/`.
29. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of OSDI'2002*, 2002.
30. G. J. Myers. *The Art of Software Testing*. Wiley, 1979.
31. G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of POPL'02 (29th ACM Symposium on Principles of Programming Languages)*, pages 128–139, Portland, January 2002.
32. J. Offutt and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.
33. C. Pasareanu, R. Pelanek, and W. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *Proceedings of CAV'2005 (17th Conference on Computer Aided Verification)*, Edinburgh, July 2005.
34. S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In *Proceedings of PLDI'2004 (ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation)*, Washington D.C., June 2004.
35. K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.
36. Simplify. web page: `http://research.compaq.com/SRC/esc/Simplify.html`.
37. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
38. Valgrind. web page: `http://valgrind.org/`.
39. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.
40. W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.
41. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Proceedings of TACAS'05 (11th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005.