Combining Model Checking and Testing

Patrice Godefroid and Koushik Sen

Abstract Model checking and testing have a lot in common. Over the last two decades, significant progress has been made on how to broaden the scope of model checking from finite-state abstractions to actual software implementations. One way to do this consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software. This chapter presents an overview of this strand of software model checking.

1 Introduction

Model checking and testing have a lot in common. In practice, the main value of both is *to find bugs* in programs. And, if no bugs are to be found, both techniques increase the confidence that the program is correct.

In theory, model checking is a form of formal verification based on exhaustive state-space exploration. As famously stated by Dijkstra decades ago, *"testing can only find bugs, not prove their absence"*. In contrast, verification (including exhaustive testing) can prove the absence of bugs. This is the key feature that distinguishes verification, including model checking, from testing.

In practice, however, the verification guarantees provided by model checking are often limited: model checking checks only a program, or a manually-written model of a program, for some specific properties, under some specific environment assumptions, and the checking itself is usually approximate for nontrivial programs and properties when an exact answer is too expensive to compute. Therefore, model checking should be viewed in practice more as a form of *"super testing"* rather

Patrice Godefroid
Microsoft Research, e-mail: `pg@microsoft.com`

Koushik Sen
UC Berkeley, e-mail: `ksen@cs.berkeley.edu`

$$\begin{tikzcd}
\text{Modeling languages} \arrow[r, "\text{state-space exploration}"] & \text{Model checking} \arrow[d, "\text{adaptation}"] \\
\text{Programming languages} \arrow[u, "\text{abstraction}"] \arrow[r, "\text{state-space exploration}"] & \textbf{Systematic testing}
\end{tikzcd}$$

Fig. 1 Two main approaches to software model checking.

than as a form of formal verification in the strict mathematical sense. Compared to testing, model checking provides better coverage, but is more computationally expensive. Compared to more general forms of program verification like interactive theorem proving, model checking provides more limited verification guarantees, but is cheaper due to its higher level of automation. Model checking thus offers an *attractive practical trade-off* between testing and formal verification.

The key practical strength of model checking is that it is able to *find bugs* that would be extremely hard to find (and reproduce) with traditional testing. This key strength has been consistently demonstrated, over and over again, during the last three decades when applying model checking tools to check the correctness of hardware and software designs, and more recently software implementations. It also explains the gradual adoption of model checking in various industrial environments over the last 20 years (hardware industry, safety-critical systems, software industry).

What prevents an even wider adoption of model checking is its relative *higher cost* compared to basic testing. This is why model checking has been adopted so far mostly in *niche* yet *critical* application domains where the cost of bugs is high enough to justify the cost of using model checking (hardware designs, communication switches, embedded systems, operating-system device drivers, security bugs, etc.).

Over the last two decades, significant progress has been made on how to lower the cost of adoption of model checking even further when applied to software through the advent of *software model checking*. Unlike traditional model checking, a software model checker does not require a user to manually write an abstract *model* of the software program to be checked in some modeling language, but instead works directly on a program *implementation* written in a full-fledged programming language.

As illustrated in Figure 1, there are essentially *two main approaches to software model checking*, i.e., two ways to broaden the scope of model checking from modeling languages to programming languages. One approach uses *abstraction*: it consists of automatically extracting an abstract model out of a software application by statically analyzing its code, and then of analyzing this model using traditional model-checking algorithms (e.g., [4, 43, 110, 84]). Another approach uses *adaptation*: it consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software (e.g., [59, 148, 107, 67]).

The aim of this chapter is to present an overview of this second approach to software model checking. We describe the main ideas and techniques used to systematically test and explore the state spaces of concurrent (Section 2) or data-driven (Section 3) software, or both (Section 4). We also discuss other related work (Section 5), such as combining systematic testing with static program analysis, run-time verification, and other testing techniques. However, this chapter is only meant to provide an introduction to this research area, *not* an exhaustive survey.

2 Systematic Testing of Concurrent Software

In this section, we present techniques inspired by model checking for systematically testing concurrent software. We discuss nondeterminism due to concurrency before nondeterminism due to data inputs (in the next section) for historic reasons. Indeed, model checking was first conceived for reasoning about concurrent reactive systems [36, 123], and software model checking via systematic testing was also first proposed for concurrent programs [59].

2.1 Classical Model Checking

Traditional model checking checks properties of a system modeled in some modeling language, typically some kind of notation for communicating finite-state machines. Given such a system's model, the formal semantics of the modeling language defines the *state space* of the model typically as some kind of *product* of the communicating finite-state machines modeling the system's components. A state space is usually defined as a directed graph whose nodes are states of the entire system and edges represent state changes. Branching in the graph represents either branching in individual state machine components or nondeterminism due to concurrency, i.e., different orderings of actions performed by different components. The state space of a system's model thus represents the joint dynamic behavior of all components interacting with each other in all possible ways. By systematically exploring its state space, model checking can reveal unexpected possible interactions between components of the system's model, and hence reveal potential flaws in the actual system.

Many properties of a system's model can be checked by exploring its state space: one can detect deadlocks, dead code, violations of user-specified assertions, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last three decades thanks to the development of model-checking methods for various temporal logics (e.g., [38, 98, 123, 145]). Historically, the term "model checking" was introduced to mean "check whether a system is a model of a temporal logic formula", in the classic logical sense. This definition does not imply that a "model", i.e., an abstraction, of a system is checked. In this chapter, we will use the term "model checking" in a broad sense, to denote any systematic state-space exploration technique that can be used for verification purposes when it is exhaustive.

2.2 Software Model Checking Using a Dynamic Semantics

Like a traditional model checker explores the state space of a system modeled as the product of concurrent finite-state components, one can systematically explore the *"product"* of concurrently executing *operating-system processes* by using a *run-time scheduler* for driving the entire software application through the states and transitions of its state space [59].

The product, or *state space*, of concurrently executing processes can be defined *dynamically* as follows. Consider a concurrent system composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of *operations* described in a sequential program written in any full-fledged programming language (such as C, C++, etc.). Such sequential programs are *deterministic*: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing *atomic operations* on communication objects. Examples of communication objects are shared variables, semaphores, and FIFO buffers. Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed; for instance, waiting for the reception of a message blocks until a message is received. We assume that only executions of visible operations may be blocking.

At any time, the concurrent system is said to be in a *state*. The system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. Every process in the system is always expected to eventually attempt executing a visible operation.[1] This assumption implies that initially, after the creation of all the processes of the system, the system can reach a first and unique global state s_0, called the *initial global state* of the system.

A *process transition*, or *transition* for short, is defined as one visible operation followed by a *finite* sequence of invisible operations performed by a *single* process and ending just before a visible operation. Let T denote the set of all transitions of the system.

A transition is said to be *disabled* in a global state s when the execution of its visible operation is blocking in s. Otherwise, the transition is said to be *enabled* in

[1] If a process does not attempt to execute a visible operation within a given amount of time, an error is reported at run-time.

s. A transition t enabled in a global state s can be *executed* from s. Since the number of invisible operations in a transition is finite, the execution of an enabled transition always terminates. When the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t and denoted by $s \xrightarrow{t} s'$.[2]

We can now define the *state space* of a concurrent system satisfying our assumptions as the transition system $A_G = (S, \Delta, s_0)$ representing its set of reachable global states and the (process) transitions that are possible between these:

- S is the set of global states of the system,
- $\Delta \subseteq S \times S$ is the *transition relation* defined as follows:

$$(s, s') \in \Delta \text{ iff } \exists t \in T : s \xrightarrow{t} s',$$

- s_0 is the initial global state of the system.

We emphasize that an element of Δ, or *state-space transition*, corresponds to the execution of a single process transition $t \in T$ of the system. Remember that we use here the term "transition" to refer to a process transition, not to a state-space transition. Note how (process) transitions are defined as maximal sequences of interprocess "local" operations from a visible operation to the next. Interleavings of those local operations are not considered as part of the state space.

It can be proved [59] that, for any concurrent system satisfying the above assumptions, exploring only all its global states is sufficient to detect all its *deadlocks* and *assertion violations*, i.e., exploring all its non-global states is not necessary. This result justifies the choice of the specific dynamic semantics described in this section. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Deadlocks are a notorious problem in concurrent systems, and can be difficult to detect through conventional testing. Assertions can be specified by the user in the code of any process with the special visible operation "assert". It takes as its argument a boolean expression that can test and compare the value of variables and data structures *local* to the process. Many undesirable system properties, such as unexpected message receptions, buffer overflows and application-specific error conditions, can easily be expressed as assertion violations.

Note that we consider here *closed* concurrent systems, where the environment of one process is formed by the other processes in the system. This implies that, in the case of a single "open" reactive system, the environment in which this system operates has to be represented somehow, possibly using other processes. In practice, a complete representation of such an environment may not be available, or may be very complex. It is then convenient to use a simplified representation of the environment, or *test driver* or *mock-up*, to simulate its behavior. For this purpose, it is useful to introduce a special operation to express a valuable feature of modeling languages, not found in programming languages: *nondeterminism*. This operation, let us call it

[2] Operations on objects (and hence transitions) are deterministic: the execution of a transition t in a state s leads to a *unique* successor state.

```
/* phil.c : dining philosophers (version without loops) */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define N 2

philosopher(i,semid)
     int i, semid;
{
  printf("philosopher %d thinks\n",i);
  semwait(semid,i,1);          /* take left fork */
  semwait(semid,(i+1)%N,1);    /* take right fork */
  printf("philosopher %d eats\n",i);
  semsignal(semid,i,1);       /* release left fork */
  semsignal(semid,(i+1)%N,1); /* release right fork */
  exit(0);
}

main()
{
  int semid, i, pid;

  semid = semget(IPC_PRIVATE,N,0600);

  for(i=0;i<N;i++)
    semsetval(semid,i,1);
  for(i=0;i<(N-1);i++) {
    if((pid=fork()) == 0)
      philosopher(i,semid);
    };
  philosopher(i,semid);
}
```

Fig. 2 Example of concurrent C program simulating dining philosophers.

`nondet`[3], takes as argument a positive integer n, and returns an integer in $[0,n]$. The operation is visible and nondeterministic: the execution of a transition starting with `nondet(`n`)` may yield up to $n+1$ different successor states, corresponding to different values returned by `nondet`. This operation can be used to represent *input data nondeterminism* or the effect of input data on the control flow of a test driver. How to deal with input data nondeterminism will be discussed further in Section 3.

Example 1. [59] Consider the concurrent C program shown in Figure 2. This program represents a concurrent system composed of two processes that communicate using semaphores. The program describes the behavior of these processes as well as the initialization of the system. This example is inspired by the well-known dining-philosophers problem, with two philosophers. The two processes communicate by executing the (visible) operations *semwait* and *semsignal* on two semaphores that are identified by the integers 0 and 1 respectively. The operations *semwait* and *semsignal* take 3 arguments: the first argument is an identifier for an array of semaphores, the second is the index of a particular semaphore in that array, and the

[3] This operation is called `VS_toss` in [59].

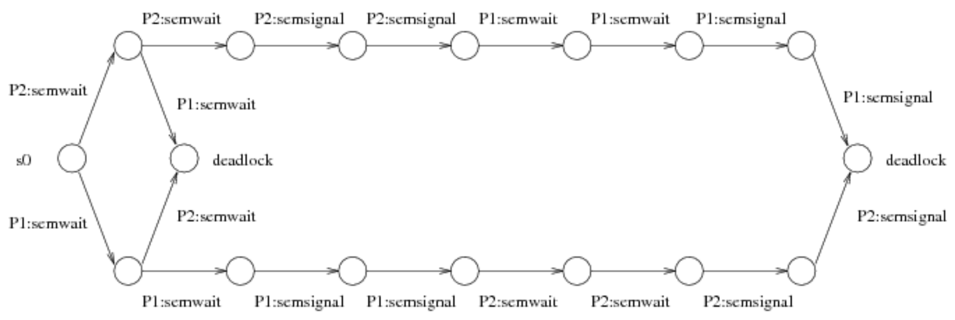

Fig. 3 Global state space for the two-dining-philosophers system.

third argument is a value by which the counter associated with the semaphore identified by the first two arguments must be decremented (in the case of *semwait*) or incremented (in the case of *semsignal*). The value of both semaphores is initialized to 1 with the operation *semsetval*. By implementing these operations using actual operating-system semaphores (for instance, the exact UNIX system calls to do this are similar), the program above can be compiled and executed. The state space A_G of this system is shown in Figure 3, where the two processes are denoted by $P1$ and $P2$, and state-space transitions are labeled with the visible operation of the corresponding process transition. The operation *exit* is a visible operation whose execution is always blocking. Since all the processes are deterministic, nondeterminism (i.e., branching) in A_G is caused only by concurrency. This state space contains two deadlocks (i.e., states with no outgoing transitions). The deadlock on the right represents normal termination (where both process are blocked on *exit*), while the deadlock on the left is due to a coordination problem.

2.3 Systematic Testing with a Run-Time Scheduler

The state space of a concurrent system as defined in the previous section can be systematically explored with a *run-time scheduler*. This scheduler controls and observes the execution of all the visible operations of the concurrent processes of the system (see Figure 4). Every process of the concurrent system to be analyzed is mapped to an operating-system process. Their execution is controlled by the scheduler, which is another process external to the system. The scheduler observes the visible operations executed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition in the state space A_G of the concurrent system.

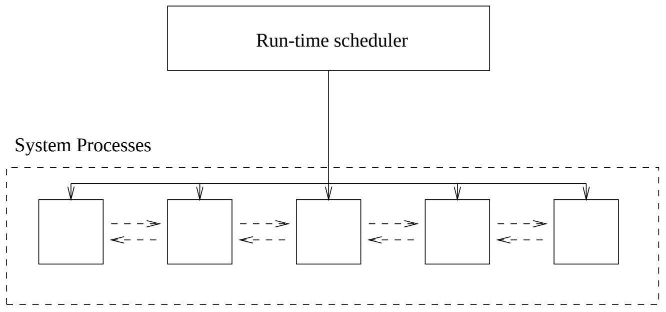

Fig. 4 Overall architecture of a dynamic software model checker for concurrent systems.

Combined with a systematic state-space search algorithm, the run-time scheduler can drive an entire application through all (or many) possible concurrent executions by systematically scheduling all possible interleavings of their communication operations. In order to explore an alternative execution, i.e., to "backtrack" in its search, the run-time scheduler can, for instance, restart the execution of the entire software application in its initial state, and then drive its execution along a different path in its state space.

Whenever an error (such as a deadlock or an assertion violation) is detected during the search, a whole-system execution defined by the sequence of transitions that leads to the error state from the initial state can be exhibited to the user. Dynamic model checkers typically also include an interactive graphical simulator/debugger for replaying executions and following their steps at the instruction or procedure/-function level. Values of variables of each process can be examined interactively. The user can also explore interactively any path in the state space of the system with the same set of debugging tools (e.g., see [60]).

As described in the previous section, we assume that there are exactly two sources of nondeterminism in the concurrent systems considered here: (1) concurrency and (2) calls to the special visible operation `nondet` used to model nondeterminism and whose return values are controlled by the run-time scheduler. When this assumption is satisfied, the run-time scheduler has complete control over nondeterminism. It can thus reproduce any execution leading to an error found during a state-space search.

Remember that the ability to provide state-space coverage guarantees, even limited ones, is precisely what distinguishes verification, including model checking, from traditional testing, as explained earlier in the introduction. This is why the term "software model checking" was applied to this approach of systematic testing with a run-time scheduler, since eventually it does provide full state space coverage.

Of course, in practice, state spaces can be huge, even infinite. But even then, the state space can always be explored exhaustively *up to some depth*, which can be increased progressively during state-space exploration using an "iterative deepening" search strategy. Efficient search algorithms, based on partial-order reduction, have been proposed for exhaustively exploring the state spaces of *message-passing* concurrent systems up to a “reasonable” depth, say, all executions with up to 50 message exchanges. In practice, such depths are often sufficient to thoroughly exercise implementations of communication protocols and other distributed algorithms. Indeed, exchanging a message is an expensive operation, and most protocols are therefore designed so that few messages are sufficient to exercise most of their functionality. By being able to systematically explore all possible interactions of the implementation of all communicating protocol entities up to tens of message exchanges, this approach to software model checking has repeatedly been proven to be effective in revealing subtle concurrency-related bugs [60].

2.4 Stateless Vs. Stateful Search

This approach to software model checking for concurrent programs thus adapts model checking into a form of systematic testing that simulates the effect of model checking while being applicable to concurrent processes executing arbitrary code written in full-fledged programming languages (like C, C++, Java, etc.). The only main requirement is that the run-time scheduler must be able to trap operating system calls related to communication (such as sending or receiving messages) and be able to suspend and resume their executions, hence effectively controlling the scheduling of all processes whenever they attempt to communicate with each other.

This approach to software model checking was pioneered in the VeriSoft tool [59]. Because states of implementations of large concurrent software systems can require megabytes or more *each* to be represented, VeriSoft does not store states in memory and simply traverse state-space paths in a *stateless* manner, exactly as in traditional testing. It is shown in [59] that in order to make a systematic stateless search tractable, partial-order reduction is necessary to avoid re-exploring over and over again parts of the state space reachable by different interleavings of a same concurrent partial-order execution.

However, for small to medium-size applications, computing state representations and storing visited states in memory can be tractable, possibly using approximations and especially if the entire state of the operating-system can be determined as is the case when the operating system is a *virtual machine*. This extension was first proposed in the Java PathFinder tool [148]. This approach limits the size and types of (here Java) programs that can be analyzed, but allows the use of standard model-checking techniques for dealing with state explosion, such as bitstate hashing, stateful partial-order reduction, symmetry reduction, and the use of abstraction techniques.

Another trade-off is to store only *partial* state representations, such as storing a hash of a part of each visited state, possibly specified by the user, as explored in the CMC tool [107]. Full state-space coverage with respect to a dynamic semantics defined at the level of operating-system processes can then no longer be guaranteed, even up to some depth, but previously visited partial states can now be detected, and multiple explorations of their successor states can be avoided, which helps focus the remainder of search on other parts of the state space more likely to contain bugs.

2.5 Systematic Testing for Multi-Threaded Programs

Software model checking via systematic testing is effective for message-passing programs because systematically exploring their state spaces up to tens of message exchanges typically exercises a lot of their functionality. In contrast, this approach is more problematic for *shared-memory* programs, such as multi-threaded programs where concurrent threads communicate by reading and writing shared variables. Instead of a few well-identifiable message queues, shared-memory communication may involve thousands of communicating objects (e.g., memory addresses shared by different threads) that are hard to identify. Moreover, while systematically exploring all possible executions up to, say, 50 message exchanges can typically cover a large part of the functionality of a protocol implementation, systematically exploring all possible executions up to 50 read/write operations in a multi-threaded program typically covers only a tiny fraction of the program functionality. How to effectively perform software model checking via systematic testing for shared-memory systems is a harder problem and has been the topic of recent research.

Dynamic partial-order reduction (DPOR) [56] dynamically tracks interactions between concurrently-executing threads in order to identify when communication takes place through which shared variables (memory locations). Then, DPOR computes backtracking points where alternative paths in the state space need to be explored because they might lead to other executions that are not "equivalent" to the current one (i.e., are not linearizations of the same partial-order execution). In contrast, traditional partial-order reduction [143, 117, 58] for shared-memory programs would require a static alias analysis to determine which threads may access which shared variables, which is hard to compute accurately and cheaply for programs with pointers. DPOR has been extended and implemented in several recent tools [150, 108, 81, 135].

Even with DPOR, state explosion is often still problematic. Another recent approach is to use *iterative context bounding*, a novel search ordering heuristics which explores executions with at most k context switches, where k is a parameter that is iteratively increased [121]. The intuition behind this search heuristics is that many concurrency-related bugs in multi-threaded programs seem due to just a few unexpected context switches. This search strategy was first implemented in the Chess tool [108].

Even when prioritizing the search with aggressive context bounding, state explosion can still be brutal in large shared-memory multi-threaded programs. Other search heuristics for concurrency have been proposed, which we could call collectively *concurrency fuzzing* techniques [50, 129, 105]. The idea is to use a random run-time scheduler that occasionally preempts concurrent executions selectively in order to increase the likelihood of triggering a concurrency-related bug in the program being tested. For instance, the execution of a memory allocation, such as `ptr=malloc(...)`, in one thread could be delayed as much as possible to see if other threads may attempt to dereference that address `ptr` before it is allocated. Unlike DPOR or context bounding, these heuristic techniques do not provide any state-space coverage guarantees, but can still be effective in practice in finding concurrency-related bugs.

Other recent work investigates the use of concurrency-related search heuristics with *probabilistic guarantees* (e.g., see [105]). This line of work attempts to develop randomized algorithms for concurrent system verification which can provide probabilistic coverage guarantees, under specific assumptions about the concurrent program being tested and for specific classes of bugs.

Active testing is a relatively new scalable automated method for directed testing of concurrent programs. Active testing combines the power of imprecise program analysis with the precision of software testing to quickly discover concurrency bugs and to reproduce discovered bugs on demand [132, 114, 90, 88, 23, 89, 87, 21, 24, 25, 22, 115, 116]. The key idea behind active testing is to control the thread scheduler in order to force the program into a state to expose a concurrency bug, e.g. data race, deadlock, atomicity violation, or violation of sequential memory consistency. The technique starts with lightweight inexpensive dynamic analysis that identifies situations where there is suspicion that a concurrency bug may exist. This first part of the analysis is imprecise because it trades-off precision for efficiency and it tries to increase the coverage of analysis by trying to predict potential bugs in other executions by analyzing a single execution. In the second step, a directed tester executes the program under a controlled thread schedule in an attempt to bring the program in the buggy state. If it succeeds, it has identified a real concurrency bug; that is, the error report is guaranteed not to be a false alarm, which is a serious problem with existing dynamic analyses. The actual method of controlling the thread schedule works as follows: once a thread reaches a state that resembles the desired state, it is paused as long as possible, giving a chance to other threads to catch up and complete the candidate buggy scenario.

2.6 Tools and Applications

We list here several tools and applications of software model checking via systematic testing for concurrent systems.

As mentioned before, the idea of dynamic software model checking via systematic testing was first proposed and implemented in the VeriSoft tool [59], developed at Bell Labs and publicly available since 1999. It has been used to find several complex errors in industrial communication software, ranging from small critical components of phone-switching software [63] to large call-processing applications running on wireless base-stations [32].

Java PathFinder [148] is another early and influential tool which analyzes Java concurrent programs using a modified Java virtual machine. It also implements a blend of several static and dynamic program analysis techniques. It has been used to find subtle errors in several complex Java components developed at NASA [16, 118]. It is currently available as an extensible open-source tool. It has been recently extended to also include test-generation techniques based on symbolic execution [2], which will be discussed in the next section.

CMC [107] analyzes concurrent C programs. It has been used to find errors in implementations of network protocols [106] and file systems [154].

jCUTE [135] is a tool for analyzing concurrent Java programs. It uses a variant of DPOR for data race detection. It also implements test-generation techniques based on concolic testing discussed in the next section.

Chess [108] analyzes multi-threaded Windows programs. It has been used to find many errors in a broad range of applications inside Microsoft [109]. It is also publicly available.

MaceMC [94] analyzes distributed systems implemented in Mace, a domain-specific language built on top of C++. This tool also specializes in finding liveness-related bugs.

MoDist [153] analyzes concurrent and distributed programs; it found many bugs in several distributed system implementations. Cuzz [105] analyzes multi-threaded Windows programs using concurrency fuzzing techniques (see previous section). ISP [150] analyzes concurrent MPI programs using stateful variants of DPOR and other techniques.

Calfuzzer [88] is an extensible and publicly available active testing tool for Java. Thrille for C/PThreads [87] and UPC-Thrille for UPC [115, 116] are active testing tools developed for C programs. These tools have been applied to find many previously-known and unknown concurrency bugs in a number of programs, including several real-world applications with millions lines of code. UPC-Thrille is an active testing tool for distributed-memory parallel programs. UPC-Thrille has been shown to scale to thousands of nodes with a maximum overhead of 50%.

3 Systematic Testing of Sequential Software

In this section, we present techniques inspired by model checking for systematically testing sequential software. We assume that nondeterminism in such programs is exclusively due to data inputs.

Enumerating all possible data inputs values with a `nondet` operation as described in Section 2.2 is tractable only when sets of possible input values are small, like selecting one choice in a menu with (few) options. For dealing with large sets of possible input data values, the main technical tool used is *symbolic execution*, which computes *equivalence classes of concrete input values* that lead to the execution of the same program path. We start with a brief overview of “classical” symbolic execution in the next section, and then describe recent extensions for systematic software testing.

3.1 Classical Symbolic Execution

Symbolic execution is a program analysis technique that was introduced in the 70s (e.g., see [95, 15, 39, 124, 85]). Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Symbolic execution can be used for many purposes, such as bug detection, program verification, debugging, maintenance, and fault localization [40].

One of the earliest proposals for using static analysis as a kind of systematic symbolic program testing method was proposed by King almost 35 years ago [95]. The idea is to symbolically explore the tree of all computations the program 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. 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 characterize the inputs that drive the program through ρ. This characterization is exact provided symbolic execution has perfect precision. 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. How to perform symbolic execution and generate path constraints is illustrated with an example later in Section 3.4.

A prototype of this system allowed the programmer to be presented with feasible paths and to experiment, possibly interactively [77], with assertions in order to force new and perhaps unexpected paths. 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/Hoare's verification method [57, 82], including inductive invariants for programs that contain loops. Since then, this line of work has been developed further in various ways, leading to various approaches of program verification, such as *verification-condition generation* (e.g., [47, 5]), *symbolic model checking* [17] and *bounded model checking* [35].

Symbolic execution is also a key ingredient for *precise test input generation* and systematic testing of *data-driven programs*. While program verification aims at proving the absence of program errors, test generation aims at generating concrete test inputs that can drive the program to execute specific program statements or paths. Work on automatic code-driven test generation using symbolic execution can roughly be partitioned into two groups: *static* versus *dynamic* test generation.

3.2 Static Test Generation

Static test generation (e.g., [95]) consists of analyzing a program *P* statically, by using symbolic execution techniques to attempt to compute inputs to drive *P* along specific execution paths or branches, *without ever executing the program*.

Unfortunately, this approach is ineffective whenever the program contains statements involving constraints outside the scope of reasoning of the theorem prover, i.e., statements "that cannot be reasoned about symbolically". This limitation is illustrated by the following example [61]:

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

Assume the constraint solver cannot "symbolically reason" about the function `hash` (perhaps because it is too complex or simply because its code is not available). This means that the constraint solver cannot generate two values for inputs `x` and `y` that are guaranteed to satisfy (or violate) the constraint `x == hash(y)`. In this case, static test generation cannot generate test inputs to drive the execution of the program `obscure` through either branch of the conditional statement: static test generation is *helpless* for a program like this. Note that, for test generation, it is not sufficient to know that the constraint `x == hash(y)` is satisfiable for *some* values of `x` and `y`, it is also necessary to generate *specific values* for `x` and `y` that satisfy or violate this constraint.

The practical implication of this fundamental limitation is significant: static test generation is doomed to perform poorly whenever precise symbolic execution is not possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

3.3 Dynamic Test Generation

A second approach to test generation is *dynamic test generation* (e.g., [96, 113, 79, 67, 27]): it consists of executing the program *P*, typically starting with some random inputs, while performing symbolic execution *dynamically*, collecting symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. The conventional stance on the role of symbolic execution is thus turned upside-down: symbolic execution is now an adjunct to concrete execution.

A key observation [67] is that, with dynamic test generation, *imprecision in symbolic execution can be alleviated using concrete values and randomization*: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete values of those inputs.

Consider again the program `obscure` given above. Even though it is impossible to generate two values for inputs `x` and `y` such that the constraint `x == hash(y)` is satisfied (or violated), it is easy to generate, for a fixed value of `y`, a value of `x` that is equal to `hash(y)` since the latter can be observed and known at run-time. By picking randomly and then fixing the value of `y`, we can first run the program, observe the concrete value c of `hash(y)` for that fixed value of `y` in that run; then, in the next run, we can set the value of the other input `x` either to c or to another value, while leaving the value of `y` unchanged, in order to force the execution of the `then` or `else` branches, respectively, of the conditional statement in the function `obscure`. (The algorithm presented in the next section does all this automatically [67].)

In other words, static test generation is unable to generate test inputs to control the execution of the program `obscure`, while dynamic test generation can *easily* drive the executions of that same program through all its feasible program paths, finding the `abort()` with no false alarms. In realistic programs, imprecision in symbolic execution typically creeps in in many places, and dynamic test generation allows test generation to recover from that imprecision. Dynamic test generation can be viewed as extending static test generation with additional run-time information, and is therefore more general, precise, and powerful.

How much more precise is dynamic test generation compared to static test generation? In [62], it is shown exactly when the "concretization trick" used in the above `obscure` example helps, and when it does not help. It is also shown that the main property of dynamic test generation that makes it more powerful than static test generation is *only* its ability to observe concrete values and to record those in path constraints. In contrast, the process of simplifying complex symbolic expressions using concrete run-time values can be accurately simulated statically using *uninterpreted functions*. However, those concrete values are necessary to effectively compute new input vectors, a fundamental requirement in test generation [62].

In principle, static test generation can be extended to concretize symbolic values whenever static symbolic execution becomes imprecise [93]. In practice, this is problematic and expensive because this approach not only requires to detect *all* sources of imprecision, but also requires one call to the constraint solver for each concretization to ensure that every synthesized concrete value satisfies prior symbolic constraints along the current program path. In contrast, dynamic test generation avoids these two limitations by leveraging a specific concrete execution as an automatic fall back for symbolic execution [67].

In summary, dynamic test generation is the *most precise* form of code-driven test generation that is known today. It is more precise than static test generation and other forms of test generation such as random, taint-based and coverage-heuristic-based test generation. It is also the most sophisticated, requiring the use of automated theorem proving for solving path constraints. This machinery is more complex and heavy-weight, but may exercise more paths, find more bugs and generate fewer redundant tests covering the same path. Whether this better precision is worth the trouble depends on the application domain.

3.4 Systematic Dynamic Test Generation

Dynamic test generation was discussed in the 90s (e.g., [96, 113, 79]) in a *property-guided* setting, where the goal is to execute a given *specific target* program branch or statement. More recently, new variants of dynamic test generation [67, 27] blend it with model checking techniques with the goal of systematically executing *all* feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, Valgrind or AppVerifier, for instance). In other words, each new input vector attempts to force the execution of the program through *some* new path, but the whole search is *not* guided by one specific target program branch or statement. By repeating this process, such a systematic search attempts to force the program to sweep through all its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [59] as presented in Section 2. Along each execution, a run-time checker is used to detect various types of errors (buffer overflows, uninitialized variables, memory leaks, etc.).

Systematic dynamic test generation as described above was introduced first in [67], as a part of an algorithm for “Directed Automated Random Testing”, or DART for short, and is also referred to as “concolic testing” [137], or “dynamic symbolic execution” [142]. Independently, [27] proposed “Execution Generated Tests” as a test generation technique very similar to DART. Also independently, [151] described a prototype tool which shares some of the same features. Subsequently, this approach was adopted and implemented in many other tools (see Section 3.6 and surveys [29, 30]).

Systematic dynamic test generation consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables v and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively [67]. A *symbolic value* is any expression e in some theory[4] $\mathscr{T}$ where all free variables are exclusively input parameters. For any program variable v, $M(v)$ denotes the *concrete value* of v in M, while $S(v)$ denotes the *symbolic value* of v in S. For notational convenience, we assume that $S(v)$ is always defined and is simply $M(v)$ by default if no symbolic expression in terms of inputs is associated with v in S. When $S(v)$ is different from $M(v)$, we say that that program variable v has a *symbolic value*, meaning that the value of program variable v is a function of some input(s) which is represented by the symbolic expression $S(v)$ associated with v in the symbolic store.

[4] A theory is a set of logic formulas.

A program manipulates the memory (concrete and symbolic stores) through statements, or *commands*, that are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $v := e$ where v is a program variable and e is an expression, a *conditional statement* of the form `if` b `then` C' `else` C'' where b denotes a boolean expression, and C' and C'' denote the unique[5] next command to be evaluated when b holds or does not hold, respectively, or `stop` corresponding to a program error or normal termination.

Given an input vector assigning a concrete value to every input parameter I_i, the program executes a unique finite[6] sequence of commands. For a finite sequence ρ of commands (i.e., a control path ρ), a *path constraint* ϕ_ρ is a *quantifier-free first-order logic formula* over theory $\mathscr{T}$ that is meant to characterize the input assignments for which the program executes along ρ. The path constraint is *sound and complete* when this characterization is exact.

A path constraint is generated during dynamic symbolic execution by collecting input constraints at conditional statements. Initially, the path constraint ϕ_ρ is defined to *true*, and the initial symbolic store S_0 maps every program variable v whose initial value is a program input: for all those, we have $S_0(v) = x_i$ where x_i is the symbolic variable corresponding to the input parameter I_i. During dynamic symbolic execution, whenever an assignment statement $v := e$ is executed, the symbolic store is updated so that $S(v) = \sigma(e)$ where $\sigma(e)$ denotes either an expression in $\mathscr{T}$ representing e as a function of its symbolic arguments, or is simply the current concrete value $M(v)$ of v if e does not have symbolic arguments or if e cannot be represented by an expression in $\mathscr{T}$. Whenever a conditional statement `if` b `then` C' `else` C'' is executed and the `then` (respectively `else`) branch is taken, the current path constraint ϕ_ρ is updated to become $\phi_\rho \wedge c$ (respectively $\phi_\rho \wedge \neg c$) where $c = \sigma(b)$. Note that, by construction, all symbolic variables ever appearing in ϕ_ρ are variables x_i corresponding to whole-program inputs I_i.

Given a path constraint $\phi_\rho = \bigwedge_{1 \leq i \leq n} c_i$, new alternate path constraints ϕ'_ρ can be defined by negating one of the constraints c_i and putting it in a conjunction with all the previous constraints: $\phi'_\rho = \neg c_i \wedge \bigwedge_{1 \leq j < i} c_j$. If path constraint generation is sound and complete, any satisfying assignment to ϕ'_ρ defines a new test input vector which will drive the execution of the program along the same control flow path up to the conditional statement corresponding to c_i where the new execution will then take the other branch. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathscr{T}$ are both *sound and complete*, that is, for all program paths ρ, the constraint solver returns a satisfying assignment for the path constraint ϕ_ρ *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). If those conditions hold, in addition to finding errors such as the reachability of bad

[5] We assume in this section that program executions are sequential and deterministic.

[6] We assume program executions terminate. In practice, a timeout can prevent non-terminating program executions and issue a run-time error.

```
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;
}
```

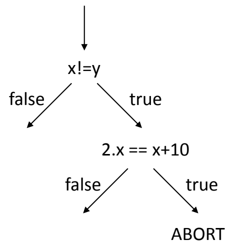

Fig. 5 A sample program (left) and the tree formed by all its path constraints (right).

program statements (like `abort()` or `assert(false)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

In practice, path constraint generation and constraint solving are usually not sound and complete. Moreover, in the presence of a single loop whose number of iterations depends on some unbounded input, the number of feasible program paths becomes infinite. In practice, search termination can always be forced by bounding input values, loop iterations or recursion, but at the cost of potentially missing bugs.

Example 2. [67] Consider the function `h` shown in Figure 5. 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 values for `x` and `y` is unlikely to discover the bug.

Assume we start with some random initial concrete input values, say `x` is initially 269167349 and `y` is 889801541. Initially, every program input is associated with a symbolic variable, denoted respectively by x and y, and every program variable storing an input value has its symbolic value (in the symbolic store) associated with the symbolic variable for the corresponding input: thus, the symbolic value for program variable `x` is the symbolic value x, and so on. Initially, the path constraint is simply $true$.

Running the function `h` with those two concrete input values 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. This execution defines a path ρ through the program. Intertwined with the normal execution, dynamic symbolic execution generate the path constraint $\phi_\rho = (x \neq y) \wedge (2 \cdot x \neq x + 10)$. Note the expression $2 \cdot x$, representing `f(x)`: it is defined through an interprocedural, dynamic tracing of symbolic expressions.

The path constraint ϕ_ρ 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 directed search generates the new path constraint, say, $\phi'_\rho = (x \neq y) \wedge (2 \cdot x = x + 10)$ obtained by negating the last constraint of the current path constraint (for instance, if the search is performed in a depth-first order). A solution to this new path constraint is $(x = 10, y = 889801541)$. A second execution of the function `h` with those two

```
struct foo {int i; char c;}
bar (char x) {
  struct foo *a;
  a->c = x;
  if (a->c == 0) {
     *((char *)a + sizeof(int)) = 1;
     if (a->c != 0)
        abort();
  }
}
```

Fig. 6 Another program example with C-like syntax.

new input values then reveals the error by driving the program into the `abort()` statement as expected.

The search space to be explored for this program is shown to the right of Figure 5. Each path in this tree corresponds to a path constraint. When symbolic execution has perfect precision as in this simple example, path constraints are both sound and complete, and dynamic and static test generation are equally powerful: they can both generate tests to drive the program along all its execution paths.

Example 3. Consider again the function `obscure`:

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

Assume we start running this program with some initial random concrete values, say `x` is initially 33 and `y` is 42. During dynamic symbolic execution, when the conditional statement is encountered, assume we do not know how to represent the expression `hash(y)`. However, we can observe dynamically that the concrete value of `hash(42)` is, say, 567. Then, the *simplified path constraint* $\phi_\rho = (x \neq 567)$ can be generated by replacing the complex/unknown symbolic expression `hash(y)` by its concrete value 567. This constraint is then negated and solved, leading to the new input vector $(x = 567, y = 42)$. Running the function `obscure` a second time with this new input vector leads to the `abort()` statement. When symbolic execution does *not* have perfect precision, dynamic test generation can be more precise than static test generation as illustrated with this example since dynamic test generation is still able to drive this program along all its feasible paths, while static test generation cannot.

Example 4. (adapted from [67]) Consider the C-like program shown in Figure 6. For this example, a static analysis will typically not be able to report with high certainty that the `abort()` is reachable. Sound static analysis tools will report “the abort might be reachable”, and unsound ones will simply report “no bug found”, if their alias analysis is not able to guarantee that `a->c` has been overwritten. In contrast, dynamic test generation easily finds a precise execution leading to the abort by simply generating an input satisfying the constraint $x = 0$. Indeed, the complex pointer arithmetic expression `*((char *)a + sizeof(int)) = 1` is *not input-dependent*, and its symbolic execution is therefore reduced to a purely concrete execution where the left-hand side of the assignment is mapped to a single concrete address – no symbolic pointer arithmetic is required, nor any pointer alias analysis. This kind of code is often found in implementations of network protocols, when a buffer of type `char *` representing an incoming message is cast into a `struct` representing the different fields of the message type.

3.5 Strengths and Limitations

At a high level, systematic dynamic test generation suffers from two main limitations:

1. the frequent imprecision of symbolic execution along individual paths, and
2. the large number of paths that usually need be explored, or *path explosion*.

In practice, however, approximate solutions to the two problems above are sufficient. To be useful, symbolic execution does not need to be perfect, it must simply be "good enough" to drive the program under test through program branches, statements and paths that would be difficult to exercise with simpler techniques like random testing. Even if a directed search cannot typically explore all the feasible paths of large programs in a reasonable amount of time, it usually does achieve better coverage than pure random testing and, hence, can find new program bugs.

Another key advantage of dynamic symbolic execution is that it can be implemented *incrementally*: only some program statements can be instrumented and interpreted symbolically, while others can simply be executed concretely natively, including all calls to external libraries and operating-system functions. A tool developer can improve the precision of symbolic execution over time, by adding new instruction handlers in a modular manner. Similarly, simple techniques like bounding the number of constraints injected at each program location are effective practical solutions to limit path explosion.

When building tools like these, there are many other challenges, which have been recently discussed in the research literature: how to recover from imprecision in symbolic execution [67, 137, 62], how to scale symbolic execution to billions of instructions [70], how to check efficiently many properties together [28, 91, 70], how to synthesize automatically symbolic instruction handlers [75], how to infer data structure invariants [92], how to reason about pointers [137, 28, 51], how to combine with random testing [100], how to find algorithmic performance problems [20], how to detect infinite loops in running programs [19], how to deal with inputs of varying sizes [152], how to deal with floating-point instructions [66], how to deal with path explosion using compositional test summaries and other caching techniques [61, 1, 13, 102, 74], which heuristics to prioritize the search in the program's search space [28, 71, 18], how to deal specifically with input-dependent loops [128, 73], how to leverage grammars (when available) for complex input formats [101, 65], how to re-use previous analysis results across code changes [119, 69, 120], how to leverage reachability facts inferred by static program analysis [74], etc. Due to space constraints, we do not discuss those challenges here, but refer instead the reader to the recent references above where those problems are discussed in detail and more pointers to other related work are provided.

3.6 Tools and Applications

Despite the limitations and challenges mentioned in the previous section, systematic dynamic test generation works well in practice: it is often able to detect bugs missed by other less precise test generation techniques. Moreover, by being grounded in concrete executions, this approach does not report false alarms, unlike traditional static program analysis. These strengths explain the popularity of the approach and its adoption in many recent tools.

Over the last several years, several tools implementing dynamic test generation have been developed for various programming languages, properties and application domains. Examples of such tools are DART [67], EGT [27], PathCrawler [151], CUTE [137], EXE [28], SAGE [71], CatchConv [104], PEX [142], KLEE [26], CREST [18], BitBlaze [139], Splat [102], Apollo [3], YOGI [74], Kudzu [127], S2E [34], CATG [130], and Jalangi [136], among others.

The above tools differ by how they perform dynamic symbolic execution (for languages such as C, Java, x86, .NET, etc.), by the type of constraints they generate (for theories such as linear arithmetic, bit-vectors, arrays, uninterpreted functions, etc.), and by the type of constraint solvers they use (such as lp_solve, CVClite, STP, Disolver, Yikes, Z3, etc.). Indeed, like in traditional static program analysis and abstract interpretation, these important parameters are determined in practice depending on which type of program is to be tested, on how the program interfaces with its environment, and on which properties are to be checked. Moreover, various cost/precision tradeoffs are also possible, as usual in program analysis.

The tools listed above also differ by the specific application domain they target, for instance protocol security [67], Unix utility programs [28, 26], database applications [52], web applications [3, 127, 136], and device drivers [74, 97]. The size of the software applications being tested also varies widely, from unit testing of programs [67, 28, 137, 18, 142, 34, 136] to system testing of very large programs with millions of lines of code [70].

At the time of this writing, the largest scale usage and deployment of systematic dynamic test generation is for *whitebox fuzzing of file parsers* [71], i.e., whole-application testing for security vulnerabilities (buffer overflows, etc.). Whitebox fuzzing scales to large file parsers embedded in applications with millions of lines of code, such as Excel, and execution traces with billions of machine instructions.

Whitebox fuzzing was first implemented in the tool SAGE [71], which uses the Z3 [46] *Satisfiability-Modulo-Theories* (SMT) solver as its constraint solver. Since 2008, SAGE has been running for over 500 machine years in Microsoft's security testing labs. This currently represents the largest computational usage for any SMT solver, with billions of constraints processed to date [14]. In the process, SAGE found new security vulnerabilities in hundreds of applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly one third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7, saving millions of dollars by avoiding expensive security patches for a billion PCs [72].

4 Systematic Testing of Concurrent Software with Data Inputs

Dynamic test generation for sequential software assumes that the program under test has no nondeterminism due to concurrency. Real-world programs have data inputs and are almost invariably concurrent. jCUTE, a dynamic test generation technique [135, 131, 134, 133], shows how to systematically test programs that have nondeterminism both due to concurrency and data inputs. The technique combines a variant of dynamic partial order reduction (see Section 2.5), called race-detection and flipping algorithm, with concolic testing. The goal of the technique is to generate thread schedules as well as data inputs that would exercise all non-equivalent executions paths of a shared-memory multi-threaded program. The technique has been implemented for Java programs in the open-source tool jCUTE (available at `http://osl.cs.illinois.edu/software/jcute/`).

jCUTE works as follows. Like DART, jCUTE executes a program both concretely and symbolically. Along the concrete execution path, jCUTE collects the constraints over the symbolic input values at each branch point and computes the path constraint. Apart from collecting symbolic constraints, jCUTE also computes race conditions (both data races and lock races) between various events in the execution of a program, where, informally, an event represents the execution of a statement in the program by a thread. We say that two events are in a *race* if they are the events of different threads, they access (i.e. read, write, lock, or unlock) the same memory location without holding a common lock, and the order of the happening of the events can be permuted by changing the schedule of the threads. The race conditions are computed by analyzing the concrete execution of the program with the help of a standard dynamic vector clock algorithm.

jCUTE first generates a random input and a schedule, which specifies the order of execution of threads. Then the algorithm does the following in a loop: it executes the code with the generated input and schedule. Along the execution path the algorithm computes the symbolic path constraint as well as the race conditions between various events. It backtracks and generates a new schedule or a new input and executes the program again. It continues until it has explored all possible distinct execution

```
                x is a shared variable
                z = input();

Thread t1                          Thread t2

1:  x = 3;                         1:  x = 2;
                                   2:  if (2*z + 1 == x)
                                   3:       abort();
```

Fig. 7 A simple shared-memory multi-threaded program *P*.

paths using a depth-first search strategy. The choice of new inputs and schedules is made in one of the following two ways:

1. The algorithm picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds, if possible, some concrete values that satisfy the new path constraint. These values are used as input for the next execution.
2. The algorithm picks two events which are in a race and generates a new schedule where the execution of the thread involved in the first event is *postponed* or *delayed* as much as possible right before the event occurs. This ensures that the events involved in the race get *flipped* or re-ordered when the program is executed with the new schedule. The new schedule is used for the next execution.

4.1 Example

We illustrate how jCUTE performs concolic testing along with race-detection and flipping using the sample program P in Figure 7. The program has two threads t_1 and t_2, a shared integer variable `x`, and an integer variable `z` which receives an input from the external environment at the beginning of the program. Each statement in the program is labeled. The program reaches the `abort()` statement in thread t_2 if the input to the program is `1` (i.e., `z` gets the value `1`) and if the program executes the statements in the following order: $(t_2,1)(t_1,1)(t_2,2)(t_2,3)$, where each event (t,l) in the sequence denotes that the thread t executes the statement labeled l.

jCUTE first generates a random input for `z` and executes P with a default schedule. Without loss of generality, the default schedule always picks the thread which is enabled and which has the lowest index. Thus the first execution of P is $(t_1,1)(t_2,1)(t_2,2)$. Let z_0 be the symbolic value of `z` at the beginning of the execution. jCUTE collects the constraints from the predicates of the branches executed in this path. For this execution, jCUTE generates the path constraint $\langle 2*z_0+1 != 2\rangle$. jCUTE also discovers that there is a race condition between the first and the second

```
t1:                 t2:                 t3:
1: x = 1;           2: y = 4;           3: x = 2;
```

Fig. 8 A three-threaded program.

event because both the events access the same variable x in different threads without holding a common lock and one of the accesses is a write of x.

Following the depth-first search strategy, jCUTE picks the only constraint $2*z_0+1!=2$, negates it, and tries to solve the negated constraint $2*z_0+1=2$. This has no solution. Therefore, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2,1)(t_2,2)(t_1,1)$. In this execution the thread involved in the first event of the race in the previous execution is delayed as much as possible. The execution thus re-orders the events involved in the race in the previous execution.

During the above execution, jCUTE generates the path constraint $\langle 2*z_0+1!=2\rangle$ and discovers that there is a race between the second and the third events. Since the negated constraint $2*z_0+1=2$ cannot be solved, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2,1)(t_1,1)(t_2,2)$. This execution re-orders the events involved in the race in the previous execution.

In the above execution, jCUTE generates the path constraint $\langle 2*z_0+1!=3\rangle$. jCUTE solves the negated constraint $2*z_0+1=3$ to obtain $z_0=1$. In the next execution, it follows the same schedule as the previous execution. However, jCUTE starts the execution with the input variable z set to 1 which is the value of z that jCUTE computed by solving the constraint. The resultant execution becomes $(t_2,1)(t_1,1)(t_2,2)(t_2,3)$ which hits the abort() statement of the program.

For each data input, the algorithm in jCUTE explores all thread schedules that are not "equivalent" to each other (i.e., are not linearizations of the same partial-order execution). Proof of this correctness result can be found in [131].

4.2 Comparison with Related Work

The race-detection and flipping algorithm developed in jCUTE is a variant of dynamic partial order reduction. The key difference between the DPOR algorithm [56] and our race-detection and flipping algorithm is that, for every choice point, the DPOR algorithm uses a persistent set while we use a postponed set [134]. These two sets can be different at a choice point. For example, for the 3-threaded program in Figure 8, if the first execution path is $(t_1,1)(t_2,2)(t_3,3)$, then at the first choice point denoting the initial state of the program, the persistent set is $\{t_1,t_3\}$; whereas, at the same choice point, the postponed set is $\{t_1\}$. (Apart from scheduling the thread t_1, the race-detection and flipping algorithm also schedules the thread t_2 at the first choice point.) Note that the DPOR algorithm picks the elements of a persistent set by using a complex forward lookup algorithm. In contrast, jCUTE *simply* puts the current scheduled thread to the postponed set at a choice point.

Note that none of the previous descriptions of DPOR techniques handled programs with data inputs. It is also worth noting that the race-detection and flipping algorithm is dynamic in nature like DPOR and addresses the limitations of static partial order reduction [143, 117, 58].

In [133] concolic testing has been extended to test asynchronous message-passing Java programs written using a Java Actor library. Shared memory systems can be modeled as asynchronous message passing systems by associating a thread with every memory location. Reads and writes of a memory location can be modeled as asynchronous messages to the thread associated with the memory location. However, this particular model would treat both reads and writes similarly. Hence, the algorithm in [133] would explore many redundant executions. For example, for the 2-threaded program $t_1 : x = 1; x = 2; t_2 : y = 3; x = 4;$, the algorithm in [133] would explore six interleavings. Our algorithm assumes that two reads are not in race and thus would explore only three interleavings of the program.

In a similar independent work [138], Siegel et al. uses a combination of symbolic execution and static partial order reduction to check if a parallel numerical program is equivalent to a simpler sequential version of the program. However, their main emphasis is in symbolic execution of numerical programs with floating points, rather than programs with pointers and data-structures. Therefore, static partial order reduction proves effective in their approach.

Model checking tools [140, 43] based on static analysis have been developed, which can detect bugs in concurrent programs. These tools employ partial order reduction techniques to reduce search space. The partial order reduction depends on detection of thread-local memory locations and patterns of lock acquisition and release.

The Path Exploration Tool (PET) [78] allows the (static) symbolic calculation of path conditions of sequential as well as concurrent systems. One can interactively change the path (for concurrent systems this can include swapping the order of concurrent transitions) to test the effect on the execution. In particular, it was shown how to limit the path condition based on a temporal property that the path needs to satisfy, represented using an automaton. In order to extend the application of symbolic testing and verification, calculating path conditions for concurrent systems with time constraints was presented in [7] and symbolic calculation of path conditions for infinite (ultimately periodic) paths was proposed in [8]. These two extensions were integrated into the PET system.

A recent related work [125] uses a set of program runs as opposed to the actual program along with concolic testing to increase coverage to concurrent program testing. Test generation is done by solving a constraint system that encodes the scheduling constraints and the data-flow constraints together. The technique is incomplete because it analyzes a subset of program traces.

More recently Farzan et al. [54] proposes an alternative technique for testing concurrent programs. The technique uses interference scenarios to systematically explore the execution space of a concurrent program. Interference scenarios capture the flow of data among different threads and enable a unified representation of path and interference constraints. The technique has been shown to scale better than jCUTE.

5 Other Related Work

The techniques we presented for software model checking by systematic testing for concurrency (Section 2) and for data inputs (Section 3) can be combined and used together (Section 4). Indeed, nondeterminism due to concurrency (whom to schedule) is *orthogonal* to nondeterminism due to input data (what values to provide). For checking most properties, concurrent programs can be sequentialized using an interleaving semantics (e.g., [59, 122]). Therefore, symbolic execution can be extended to multi-threaded programs [2, 135], since threads share the same memory address space, and take advantage of partial-order reduction (e.g., [58, 56]). The case of multi-process programs is more complicated since a good solution requires tracking symbolic variables across process boundaries and through operating systems objects such as message queues.

Static Abstraction-Based Software Model Checking. As mentioned in the introduction, essentially two approaches to software model checking have been proposed and are still actively investigated. The first approach is the one presented in the previous sections. The second 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 (if any) back to the code. The investigation of this *abstraction-based* second approach can be traced back to early attempts to analyze concurrent programs written in concurrent programming languages such as Ada (e.g., [141, 99, 103, 42]). Other relevant work includes static analyses geared towards analyzing communication patterns in concurrent programs (e.g., [41, 44, 147]). Starting in the late 90s, several efforts have aimed at providing model-checking tools based on source-code abstraction for mainstream popular programming languages such as C and Java. For instance, Feaver [84] can translate C programs into Promela, the input language of the SPIN model checker, using user-specified abstraction rules. Similarly, Bandera [43] can translate Java programs to the (finite-state) input languages of existing model checkers like SMV and SPIN, using user-guided abstraction, slicing and abstract interpretation techniques. The abstraction process can also be made *fully automatic* and adjustable depending on the specific property to be checked. For instance, SLAM [4] can translate sequential C programs to “boolean programs”, which are essentially inter-procedural control-flow graphs extended with boolean variables, using an *automatic* iterative abstraction-refinement process based on the use of predicate abstraction and a specialized model-checking procedure. For the specific classes of programs that these tools can handle, the use of abstraction techniques can produce a “conservative” model of a program that preserves basic information about the execution and communication patterns taking place in the system executing the program. Analyzing such a model using standard model-checking techniques can then prove the absence of certain types of errors in the system, without ever executing the program itself.

This second approach of static software model checking via abstraction is *complementary* to dynamic software model checking via systematic testing. Both approaches inherit the well-known advantages and limitations of, respectively, static and dynamic program analysis (e.g., [53]). Static analysis is faster than testing, provides better code coverage, but is usually less precise, is language dependent, and may produce spurious counter-examples (i.e, suffers from "false alarms/positives"). In contrast, dynamic analysis is precise, more language-independent, detects real bugs, but is slower, provides usually less coverage, and can miss bugs (i.e., suffers from "false negatives"). Overall, static and dynamic program analysis have *complementary strengths and weaknesses*, and are *worth combining*.

Combining Static and Dynamic Software Model Checking. There are many ways to combine static and dynamic program analysis, and, similarly, to combine static and dynamic software model checking. Several algorithms and tools combine static and dynamic program analyses for property checking and test generation, e.g., [111, 149, 9, 45, 11, 10, 12]. Most of these loose combinations perform a static analysis before a dynamic analysis, while some [11, 10, 12] allow for some feedback to flow between both. A tight integration between static and dynamic software model checking is proposed in a series of algorithms named Synergy [76], Dash [6] and Smash [74], and implemented in the Yogi tool [112]. The latest of these algorithms performs a compositional interprocedural may/must program analysis, where two complementary sets of techniques are used and intertwined together: a may analysis for finding proofs based on predicate abstraction and automatic abstraction refinement as in SLAM [4], and a must analysis for finding bugs based on dynamic test generation as in DART [67]. These two analyses are performed together, in coordination, and communicate their respective intermediate results to each other in the form of reusable may and must procedure summaries. This fined-grained coupling between may and must summaries allows using either type of summaries in a flexible and demand-driven manner for both proving and disproving program properties in a sound manner, and was shown experimentally to outperform previous algorithms of this kind for property-guided verification [74].

Run-Time Verification. In contrast, the approach taken in systematic dynamic test generation (see Section 3.4) is *not* property-guided: the goal is instead to exercise as many program paths as possible while checking *many* properties simultaneously along each of those paths [70]. In a non-property guided setting, the effectiveness of static analysis for safely cutting parts of the search space is typically more limited. In this context, other complementary work includes tools like Purify, Valgrind and AppVerifier, that automatically instrument code or executable files for monitoring program executions and detecting at run-time standard programming and memory-management errors such as array out-of-bounds and memory leaks. Also, several tools for so-called *run-time verification* that monitor at run-time the behavior of a reactive program and compare this behavior against an application-specific high-level specification (typically a finite-state automaton or a temporal-logic formula) have recently been developed (e.g., [49, 80]). These tools can also be used in conjunction with dynamic software model checkers.

Model-Based Testing. Software model checking via systematic testing differs from *model-based testing*. Given an abstract representation of the program, called *model*, model-based testing consists in generating tests to check the *conformance* of the program with respect to the model (e.g., [155, 48, 126, 33, 86, 144, 146]). In contrast, systematic testing does not use or require a model of the program under test. Instead, its goal is to generate tests that exercise as many program statements as possible, including assertions inserted in the code. Another fundamental difference is that models are usually written in abstract modeling languages which are, by definition, more amenable to precise analysis, symbolic execution and test generation. In contrast, code-driven test generation has to deal with arbitrary software code and systems for which program analysis is bound to be imprecise, as we discussed in Sections 3.2 and 3.3. Sometimes, the model itself is specified as a product of finite-state machines (e.g., [55]). In that case, systematic state-space exploration techniques inspired by traditional finite-state model checking are used to automatically generate a set of test sequences that cover the concurrent model according to various coverage criteria.

Must Program Abstractions. Test generation is only one way of *proving existential reachability properties* of programs, where specific concrete input values are generated to exercise specific program paths. More generally, such properties can be proved using so-called *must abstractions* of programs [64], without necessarily generating concrete tests. A must abstraction is defined as a program abstraction that preserves existential reachability properties of the program. Sound path constraints are particular cases of must abstractions [74]. Must abstractions can also be built backwards from error states using static program analysis [31, 83]. This approach can detect program locations and states provably leading to error states (no false alarms), but may fail to prove reachability of those error states back from whole-program initial states, and hence may miss bugs or report unreachable error states.

Other Verification Approaches from Programs to Logic. As mentioned earlier in Section 3.1, test generation using symbolic execution, path constraints and constraint solving is related to other approaches to program verification which reason about programs using logic. Examples of such approaches are verification-condition generation [47, 5], symbolic model checking [17] and SAT/SMT-based bounded model checking [35, 37]. All these approaches have a lot in common, yet differ by important details. In a nutshell, these approaches translate an entire program into a single logic formula using static program analysis. This logic encoding usually tracks both control and data dependencies on all program variables. Program verification is then usually reduced to a validity check using an automated theorem prover. When applicable, those approaches can efficiently prove complex properties of programs. In contrast, test generation using symbolic execution builds a logic representation of a program *incrementally*, one path at a time. Path-by-path program exploration obviously suffers from the path explosion problem discussed earlier, but it scales to large complex programs which are currently beyond the scope of applicability of other automatic program verification approaches like SAT/SMT-based bounded model checking. Verification-condition generation has been shown to scale to some large programs but it is not automatic: it typically requires a large quantity of non-trivial user-annotations, like loop invariants and function pre/post conditions, to work in practice and is more similar to interactive theorem proving. We refer the reader to [68] for a more detailed comparison of all these approaches.

6 Conclusion

We discussed how model checking can be combined with testing to define a dynamic form of software model checking based on systematic testing, which scales to industrial-size concurrent and data-driven software. This line of work was developed over the last two decades and is still an active area of research. This approach has been implemented in tens of tools by now. The application of those tools have, collectively, found thousands of new bugs, many of those critical from a reliability or security point of view, in many different application domains.

Yet much is still to be accomplished. Software model checking has been successfully applied to several niche applications, such as communication software, device drivers and file parsers, but has remained elusive for general-purpose software. Most tools mentioned in the previous sections are research prototypes, aimed at exploring new ideas, but they are not used on a regular basis by ordinary software developers and testers. Finding other "killer apps" for these techniques, beyond device drivers [4] and file parsers [71], is *critical* in order to sustain progress in this research area.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381, Budapest, April 2008. Springer-Verlag.
2. S. Anand, C. Pasareanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, pages 134–138, 2007.
3. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010.
4. 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.
5. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO'2005 (4th International Symposium on Formal Methods for Components and Objects)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, September 2006.
6. 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.
7. S. Bensalem, D. Peled, H. Qu, and S. Tripakis. Generating Path Conditions for Timed Systems. In *Proceedings of IFM'2005*, pages 5–19, 2005.
8. S. Bensalem, D. Peled, H. Qu, S. Tripakis, and L. Zuck. Test Case Generation for Ultimately Periodic Paths. In *Proceedings of HVC'2007*, pages 120–135, 2007.
9. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE 2004, Edinburgh, May 26-28)*, pages 326–335. IEEE Computer Society Press, Los Alamitos (CA), 2004.
10. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and Ph. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the 20th ACM SIGSOFT*

International Symposium on the Foundations of Software Engineering (FSE 2012, Cary, NC, November 10-17). ACM, 2012.
11. D. Beyer, T. A. Henzinger, and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. In *Proceedings of ASE'2008 (23rd International Conference on Automated Software Engineering)*, L'Aquila, September 2008.
12. D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013, Rome, Italy, March 19-22)*, LNCS 7792, pages 472–491. Springer-Verlag, Heidelberg, 2013.
13. P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, April 2008.
14. E. Bounimova, P. Godefroid, and D. Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of ICSE'2013 (35th International Conference on Software Engineering)*, pages 122–131, San Francisco, May 2013. ACM.
15. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10:234–245, 1975.
16. G. Brat, D. Drusinsky, D. Giannakopolou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, and W. Visser. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in System Design*, 25, 2004.
17. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of LICS'1990 (5th Symposium on Logic in Computer Science)*, pages 428–439, Philadelphia, June 1990.
18. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, 2008.
19. J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *24th IEEE/ACM nternational Conference on Automated Software Engineering (ASE'09)*. IEEE, 2009.
20. J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *31st International Conference on Software Engineering (ICSE'09)*, pages 463–473. IEEE, 2009.
21. J. Burnim, G. Necula, and K. Sen. Separating functional and parallel correctness using nondeterministic sequential specifications. In *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, 2010.
22. J. Burnim, G. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 79–90. ACM, 2011.
23. J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. *Communications of the ACM*, 53(6):97–105, June 2010.
24. J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *International Conference on Tools and Algorithms for the Construction ana Analysis of Systems (TACAS'11)*, pages 11–25, 2011.
25. J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, 2011.
26. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, Dec 2008.
27. 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.
28. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.
29. C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N.Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In *ICSE'2011*, Honolulu, May 2011.
30. C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

31. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of PLDI'2009 (ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation)*, Dublin, June 2009.
32. S. Chandra, P. Godefroid, and C. Palm. Software Model Checking in Practice: An Industrial Case Study. In *Proceedings of ICSE'2002 (24th International Conference on Software Engineering)*, pages 431–441, Orlando, May 2002. ACM.
33. J. Chang, D. Richardson, and S. Sankar. Structural Specification-based Testing with ADL. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 62–70, San Diego, January 1996.
34. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of ASPLOS'2011*, 2011.
35. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
36. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
37. E. M. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.
38. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.
39. L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.
40. L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.
41. C. Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, New York, NY, USA, June 1995. ACM Press.
42. J. C. Corbett. Constructing abstract models of concurrent real-time software. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 250–260, San Diego, January 1996.
43. 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.
44. R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 214–225, New York, NY, USA, June 1995. ACM Press.
45. 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.
46. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, April 2008. Springer-Verlag.
47. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
48. L.K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, 19(5):140–153, December 1994. Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering.
49. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 2000 SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer-Verlag, 2000.

50. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
51. B. Elkarablieh, P. Godefroid, and M.Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *Proceedings of ISSTA'09 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 129–139, Chicago, July 2009.
52. M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *Proceedings of ISSTA'2007 (International Symposium on Software Testing and Analysis*, pages 151–162, 2007.
53. M. D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of WODA'2003 (ICSE Workshop on Dynamic Analysis)*, Portland, May 2003.
54. A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '13)*, 2013.
55. J.-C. Fernandez, C. Jard, Th. Jeron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, August 1996. Springer-Verlag.
56. 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.
57. R. Floyd. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.
58. 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.
59. 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.
60. P. Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, March 2005. Also available as Bell Labs Technical Memorandum ITD-03-44189G, March 2003.
61. P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.
62. P. Godefroid. Higher-Order Test Generation. In *Proceedings of PLDI'2011 (ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation)*, pages 258–269, San Jose, June 2011.
63. P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ISSTA'98 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 124–133, Clearwater Beach, March 1998.
64. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *Proceedings of CONCUR'2001 (12th International Conference on Concurrency Theory)*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, Aalborg, August 2001. Springer-Verlag.
65. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of PLDI'2008 (ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation)*, pages 206–215, Tucson, June 2008.
66. P. Godefroid and J. Kinder. Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis. In *Proceedings of ISSTA'2010 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 1–11, Trento, July 2010.
67. 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.

68. P. Godefroid and S. K. Lahiri. From Program to Logic: An Introduction. In *Proceedings of the LASER'2011 Summer School*, volume 7682 of *Lecture Notes in Computer Science*, pages 31–44, Elba, December 2012. Springer-Verlag.
69. P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of SAS'2011 (18th International Static Analysis Symposium)*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128, Venice, September 2011. Springer-Verlag.
70. P. Godefroid, M.Y. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software)*, pages 207–216, Atlanta, October 2008. ACM Press.
71. P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.
72. P. Godefroid, M.Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, March 2012.
73. P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of ISSTA'2011 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 23–33, Toronto, July 2011.
74. P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010 (37th ACM Symposium on Principles of Programming Languages)*, pages 43–55, Madrid, January 2010.
75. P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of PLDI'2012 (ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation)*, pages 441–452, Beijing, June 2012.
76. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*, 2006.
77. 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.
78. E. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
79. 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.
80. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of RV'2001 (First Workshop on Runtime Verification)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, July 2001.
81. C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. In *6th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, San Jose, CA, USA, November 2006. IEEE.
82. C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
83. J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schaf, and Th. Wies. It's doomed; we can prove it. In *Proceedings of 2009 World Congress on Formal Methods*, 2009.
84. 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.
85. W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
86. L. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th IEEE International Conference on Software Engineering*, 1997.

87. N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *ACM SIGSOFT Eighteenth Symposium on the Foundations of Software Engineering (FSE'10)*. ACM, 2010.
88. P. Joshi, M. Naik, C. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 675–681. Springer, 2009.
89. P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *ACM SIGSOFT Eighteenth Symposium on the Foundations of Software Engineering (FSE'10)*. ACM, 2010.
90. P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 110–120. ACM, 2009.
91. P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: Amplifying the effectiveness of software testing (poster paper). In *6th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 561–564. ACM, 2007.
92. Y. Kannan and K. Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 283–294. ACM, 2008.
93. S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS'03*, April 2003.
94. C. E. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of NSDI'2007*, 2007.
95. J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.
96. B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.
97. V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC'10*, June 2010.
98. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.
99. D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of ACM Symposium on Testing, Analysis, and verification (TAV4)*, pages 21–35, Vancouver, October 1991.
100. R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proceedings of ICSE'2007 (29th International Conference on Software Engineering)*, Minneapolis, May 2007. ACM.
101. R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *ASE*, 2007.
102. R. Majumdar and R. Xu. Reducing test inputs using information partitions. In *CAV'09*, pages 555–569, 2009.
103. S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel programming*, pages 129–138, San Diego, May 1993.
104. D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors, 2007. UC Berkeley EECS, 2007-23.
105. M. Musuvathi, S. Burckhardt, P. Kothari, and S. Nagarakatte. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of ASPLOS'2010*, 2010.
106. M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proceedings of NSDI'2004*, pages 155–168, 2004.
107. 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.
108. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI'2007*, 2007.
109. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of OSDI'2008*, 2008.

110. K. S. Namjoshi and R. K. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *Proceedings of the 12th Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449, Chicago, July 2000. Springer-Verlag.
111. 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.
112. A. V. Nori and S. K. Rajamani. An Empirical Study of Optimizations in Yogi. In *ICSE'2010*, May 2010.
113. A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software Practice and Experience*, 29(2):167–193, 1999.
114. C. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th International Symposium on Foundations of Software Engineering (FSE'08)*, pages 135–145. ACM, 2008.
115. C. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, page 51. ACM, 2011.
116. C. Park, K. Sen, and C. Iancu. Scaling data race detection for partitioned global address space programs. In *27th International Conference on Supercomputing (ICS'13)*, pages 47–58. ACM, 2013.
117. D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, June 1993. Springer-Verlag.
118. J. Penix, W. Visser, S. Park, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger. Verifying time partiotioning in the DEOS scheduling kernel. *Formal Methods in System Design*, 26, 2005.
119. S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.
120. S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI'2011*, pages 504–515, San Jose, June 2011.
121. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *Proceedings of TACAS'05 (11th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
122. 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.
123. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.
124. C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Trans. on Software Engineering*, 2(4):293–300, 1976.
125. N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta. Concurrent test generation using concolic multi-trace analysis. In *10th Asian Symposium on Programming Languages and Systems (APLAS '12)*, pages 239–255, 2012.
126. D.J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.
127. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
128. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-Extended Symbolic Execution on Binary Programs. In *ISSTA'2009*, pages 225–236, Chicago, July 2009.
129. K. Sen. Effective random testing of concurrent programs. In *Proceedings of ASE'2007*, 2007.
130. K. Sen. Catg: A concolic testing tool for sequential java programs. https://github.com/ksen007/janala2.

131. K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, June 2006.
132. K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 11–21. ACM, 2008.
133. K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *LNCS*, pages 339–356. Springer, 2006.
134. K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.
135. K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.
136. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE'13*, August 2013.
137. 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.
138. S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, University of Massachusetts Department of Computer Science, 2005.
139. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS'2008*, December 2008.
140. S. D. Stoller. Model Checking Multi-Threaded Distributed Java Programs. In *Proceedings of SPIN'2000 (7th SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
141. R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, pages 362–376, May 1983.
142. N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.
143. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.
144. H.M. van der Bijl, A. Rensink, and G.J. Tretmans. Compositional Testing with ioco. In *Proceedings of FATES'2004 (Formal Approaches to Software Testing)*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2004.
145. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.
146. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949, pages 39–76. Springer-Verlag, 2008.
147. A. Venet. Abstract interpretation of the π-calculus. In Mads Dam, editor, *Analysis and Verification of Multiple-Agent Languages (Proceedings of the Fifth LOMAPS Workshop)*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1997.
148. 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.
149. 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.

150. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal Verification of Practical MPI Programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–269, 2009.
151. N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of EDCC'2005*, pages 281–292, Budapest, April 2005.
152. R. Xu, , P. Godefroid, and R. Majumdar. Testing for Buffer Overflows with Length Abstraction. In *Proceedings of ISSTA'08 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 27–38, Seattle, July 2008.
153. J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of NSDI'2009*, pages 213–228, 2009.
154. J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of OSDI'2004*, 2004.
155. M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 476–485, 1991.