Between Testing and Verification: Dynamic Software Model Checking

Patrice GODEFROID
Microsoft Research

Abstract. Dynamic software model checking consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software. Over the last two decades, dozens of tools following this paradigm have been developed for checking concurrent and data-driven software. Compared to traditional software testing, dynamic software model checking provides better coverage, but is more computationally expensive. Compared to more general forms of program verification like interactive theorem proving, this approach provides more limited verification guarantees, but is cheaper due to its higher level of automation. Dynamic software model checking thus offers an attractive practical trade-off between testing and formal verification. This paper presents a brief introduction to dynamic software model checking.

Keywords. Software Model Checking, Systematic Testing, Verification, Bug Finding

1. Introduction

Model Checking was introduced more than 30 years ago [15,61] as an automated verification technique for checking the correctness of concurrent reactive systems. Its basic idea is conceptually simple: when designing, implementing and testing a concurrent reactive system, check its correctness by modeling each component of the system using some form of (extended) finite-state machine, and then by systematically exploring the product of such finite-state machines, often called the *state space* of the system. The state space is a directed graph whose nodes represent states of the whole system, and whose 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 thus represents the joint behavior of all its 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.

Model checking thus means to check whether a system satisfies a property by exploring its state space. 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. In this paper, 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.

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 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 (hardware industry, safety-critical systems, software industry).

Over the last 20 years, significant progress has been made on how to apply model checking to *software*, i.e., precise descriptions of software implementations written in programming languages (like C, C++ or Java) and of realistic sizes (often hundreds of thousands lines of code or more). 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 *adaptation*: it consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software (e.g., [26,71,53,32]). Another 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., [3,18,55,44]).

The aim of this paper is to present a brief introduction to the first approach to software model checking. We discuss the main ideas and techniques used to systematically test and explore the state spaces of concurrent and data-driven software. This paper only provides a brief introduction to this research area, *not* an exhaustive survey.

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

Figure 1. Two main approaches to software model checking.

2. Dynamic Software Model Checking: Dealing with Concurrency

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 [15,61], and software model checking via systematic testing was also first proposed for concurrent programs [26].

2.1. 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 [26].

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 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 [26] 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

[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.

[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.

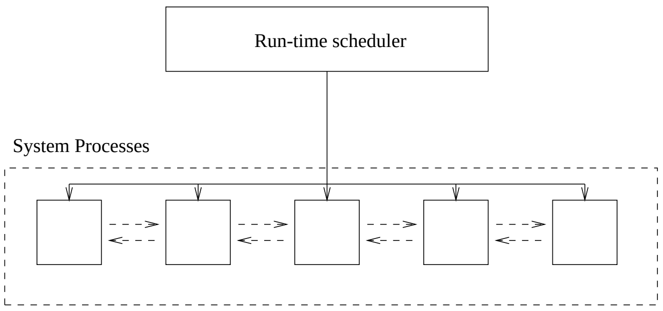

Figure 2. Overall architecture of a dynamic software model checker for concurrent systems.

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 `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.

2.2. *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 2). Every process of the concurrent system to be analyzed is mapped to an operating-system process. Their execution is controled 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.

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 lead to the error state from the initial state can be exhibited to the user. Dynamic software model

[3]This operation is called `VS_toss` in [26].

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 [27]).

It is thus assumed that there are exactly two sources of nondeterminism in the concurrent systems considered here: concurrency and calls to the special visible operation `nondet` used to model nondeterminism as described in the previous section and whose return values are controled 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. It can also guarantee, from a given initial state, *complete coverage* of the state space *up to some depth*.

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 [27].

2.3. 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 [26]. Because *each* state of implementations of large concurrent software systems can require megabytes of storage, 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 [26] 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 [71]. 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 [53]. 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.

2.4. 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) [23] 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 [70,57,25] 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 [72,54,42,66].

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 [60]. 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 [54].

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 could be called collectively *concurrency fuzzing* techniques [20,65,8]. 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 [8]). 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.

The work reported in this paper is only a partial overview of research in this area. Especially during the last decade, dozens of other tools have been developed for software model checking via systematic testing for concurrent systems, for various programming languages and application domains.

3. Dynamic Software Model Checking: Dealing with Data Inputs

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 2.1 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 [47,6,16,62,45]). 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 [17].

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 [47]. 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 [40], 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 [24,43], 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., [19,4]), *symbolic model checking* [7] and *bounded model checking* [14].

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., [47]) 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 [28]:

```
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., [48,56,41,32,11]): 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 [32] 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`.

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 [29], 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 [29].

In principle, static test generation can be extended to concretize symbolic values whenever static symbolic execution becomes imprecise [46]. 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 [32].

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., [48,56,41]) 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 [32,11] 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* [26] 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 [32], as a part of an algorithm for “Directed Automated Random Testing”, or DART for short. Independently, [11] proposed “Execution Generated Tests” as a test generation technique very similar to DART. Also independently, [73] described a prototype tool which shares some of the same features. Subsequently, this approach was adopted and implemented in many other tools (see Section 3.5), and is also sometimes casually referred to as “concolic testing” [67], or simply “dynamic symbolic execution” [69].

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 [32]. 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.

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.

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

[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.

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 $\mathcal{T}$ 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 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 1 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. ■

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* [28].

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 [32,29], how to scale symbolic execution to billions of instructions [34], how to check efficiently many properties together [12,34], how to synthesize automatically symbolic instruction handlers [39], how to precisely reason about pointers [67,12,21], how to deal with inputs of varying sizes [74], how to deal with floating-point instructions [31], how to deal with path explosion using compositional test summaries and other caching techniques [28,1,5,51,37], which heuristics to prioritize the search in the program's search space [12,35,9], how to deal specifically with input-dependent loops [64,36], how to leverage grammars (when available) for complex input formats [50,30], how to re-use previous analysis results across code changes [58,33,59], how to leverage reachability facts inferred by static program analysis [37], 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.

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 [32], EGT [11], PathCrawler [73], CUTE [67], EXE [12], SAGE [35], CatchConv [52], PEX [69], KLEE [10], CREST [9], BitBlaze [68], Splat [51], Apollo [2], YOGI [37], Kudzu [63], and S2E [13], 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 [32], Unix utility programs [12,10], database applications [22], web applications [2,63], and device drivers [37,49]. The size of the software applications being tested also varies widely, from unit testing of small programs [32,12,69,13] to system testing of very large programs with millions of lines of code [34].

4. 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. Two main techniques are used to limit state and path explosion: *partial-order reduction* when dealing with nondeterminism due to concurrency, and *dynamic symbolic execution* when dealing with nondeterminism due to data inputs. Both techniques partition the set of concrete program executions into equivalence classes in such a way that equivalent executions are indistinguishable with respect to the given properties being checked.

Dynamic software model checking was developed over the last 20 years and is still an active area of research. This approach has been implemented in dozens 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. This paper presents only a partial overview of this research area.

Acknowledgements. This paper covers two of the lectures I gave at the 2015 Marktoberdorf Summer School. I thank the organizers of the Summer School for encouraging me to write this paper. Most of the contents of this paper have appeared in various forms in several of my previous papers published during the last 20 years, such as [26,27,32,29]. A longer and more comprehensive presentation of this topic will appear soon in [38].

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. 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.

[3] 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.

[4] 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.

[5] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, April 2008.

[6] 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.

[7] 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.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of ASPLOS'2010*, 2010.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, 2008.

[10] 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.

[11] 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.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[13] 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.

[14] 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.

[15] 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.

[16] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[17] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.

[18] 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.

[19] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] R. Floyd. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incre-

mental 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] P. Godefroid and K. Sen. Combining Model Checking and Testing. In *To appear in Handbook of Model Checking*. Springer-Verlag, 2016.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[44] 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.

[45] W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[46] S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS'03*, April 2003.

[47] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[48] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[49] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC'10*, June 2010.

[50] R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *ASE*, 2007.

[51] R. Majumdar and R. Xu. Reducing test inputs using information partitions. In *CAV'09*, pages 555–569, 2009.

[52] D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors, 2007. UC Berkeley EECS, 2007-23.

[53] 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.

[54] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI'2007*, 2007.

[55] 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.

[56] 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.

[57] 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.

[58] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential Symbolic Execution. In *FSE'2008*, pages 226–237, 2008.

[59] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI'2011*, pages 504–515, San Jose, June 2011.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] K. Sen. Effective random testing of concurrent programs. In *Proceedings of ASE'2007*, 2007.

[66] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.