Software Model Checking

RANJIT JHALA
University of California, San Diego
RUPAK MAJUMDAR
University of California, Los Angeles

Software model checking is the algorithmic static analysis of programs to prove behavioral properties. It traces its roots to logic and theorem proving, both to provide the conceptual framework in which to formalize the fundamental questions and to provide algorithmic procedures for the analysis of logical questions. The undecidability theorem [Turing 1936] ruled out the possibility of a sound and complete algorithmic solution for any sufficiently powerful programming model, and even under restrictions (such as finite state spaces), the correctness problem remained computationally intractable. However, just because a problem is hard does not mean it never appears in practice. Also, just because the *general* problem is undecidable does not imply that specific *instances* of the problem will also be hard.

As the complexity of software systems grew, so did the need for some reasoning mechanism about correct behavior. (While we focus here on *analyzing* the behavior of a program relative to given correctness specifications, the development of *specification mechanisms* happened in parallel, and is covered in a different survey.)

Initially, the focus of program verification research was on manual reasoning, and the development of axiomatic semantics and logics for reasoning about programs [Floyd 1967; Hoare 1969; Dijkstra 1976; Apt and Olderog 1991] provided a means to treat programs as logical objects. As the size of software systems grew, the burden of providing entire manual proofs became too cumbersome. This marked a trend toward automating the more mundane parts, leaving the human to provide guidance to an automatic tool (for example, through loop invariants and function pre- and post-conditions [Dijkstra 1976]). This trend has continued since: the goal of software model checking research is to expand the scope of automated techniques for program reasoning, leaving less work for the expert human programmer.

More recently, software model checking has been influenced by three parallel but somewhat distinct developments. First, development of program logics and associated decision procedures [Nelson 1981; Nelson and Oppen 1980; Shostak 1984] provided a framework and basic algorithmic tools to reason about infinite state spaces. Second, automatic *model checking* techniques for temporal logics [Clarke and Emerson 1981; Queille and Sifakis 1981; Pnueli 1977; Vardi and Wolper 1994] provided basic algorithmic tools for state-space exploration. Third, compiler analysis, formalized by *abstract interpretation*, provided connections between the logical world of infinite state spaces and the algorithmic world of finite representations. Throughout the 1980s and 1990s, the three communities developed with only occasional interactions. However, in the last decade, there has been a convergence in the research directions and modern software model checkers are a culmination of ideas that combine and perhaps supersede

each area alone. In particular, the term "software model checker" is probably a misnomer, since modern tools simultaneously perform analyses traditionally classified as dataflow analysis, or theorem proving, or model checking. We retain the term solely to reflect historical development.

In this survey we trace some of these ideas that have combined to produce precise and automatic tools for the analysis of software systems.

1. PRELIMINARY DEFINITIONS

1.1 Simple Programs

We assume a transition-relation representation of programs, following the style of Manna and Pnueli [Manna and Pnueli 1992]. Over the course of this chapter, we define several classes of programs, starting with a simple model, and adding more features as we go along. To begin with, we consider *simple* programs which are defined over a set of integer variables. In the following sections, we augment this simple model with pointers and procedure calls.

A *simple imperative program* $P = (X, L, \ell_0, T)$ consists of a set X of typed variables, a set L of control locations, an initial location $\ell_0 \in L$, and a set T of transitions. Each transition $\tau \in T$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in L$ are control locations, and ρ is a constraint over free variables from $X \cup X'$. The variables from X denote values at control location ℓ, and the variables from X' denote the values of the variables from X at control location ℓ'. The sets of locations and transitions naturally define a directed labeled graph, called the *control-flow graph* (CFG) of the program. We denote by imp the class of all simple programs.

An imperative program with assignments and conditional statements can be translated into a simple program as follows. The control flow of the program is captured by the graph structure. An assignment $\mathtt{x} \leftarrow \mathtt{e}$ (where $\mathtt{e}$ is an expression over X) is translated to the relation:

$$\mathtt{x}' = \mathtt{e} \wedge \bigwedge_{\mathtt{y} \in X \setminus \{\mathtt{x}\}} \mathtt{y}' = \mathtt{y}$$

and a conditional p (a Boolean expression over X) is translated to

$$p \wedge \bigwedge_{\mathtt{x} \in X} \mathtt{x}' = \mathtt{x}$$

We shall exploit this translation to provide our examples in a C-like syntax for better readability.

Similarly, there is a simple encoding of simple programs executing concurrently with an interleaved semantics into a single simple program, for example, by taking the union of the variables, the cross product of the locations of the threads, and transitions $((l_1, l_1'), \rho, (l_2, l_2'))$ where either (l_1, ρ, l_2) is a transition of the first thread and $l_2' = l_1'$, or (l_1', ρ, l_2') is a transition of the second thread and $l_2 = l_1$.

Thus, we shall use simple programs in our exposition of model checking algorithms in the following sections. Of course, particular model checkers may have more structured representations of programs that can be exploited by the model checker. For example, input formats can include primitive synchronization operations (*e.g.,* locks, semaphores, or atomic sections), language-level support for

channels that are used for message passing, etc. In each case, such features can be compiled down to the "simple" model.

A *state* of the program P is a valuation of the variables from X. The set of all states is denoted $v.X$. We shall represent sets of states using constraints. For a constraint ρ over $X \cup X'$ and a valuation $(s, s') \in v.X \times v.X'$, we write $(s, s') \models \rho$ if the valuation satisfies the constraint ρ. A *finite computation* of the program P is a finite sequence $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots, \langle \ell_k, s_k \rangle \in (L \times v.X)^*$, where ℓ_0 is the initial location and for each $i \in \{0, \ldots, k-1\}$, there is a transition $(\ell_i, \rho, \ell_{i+1}) \in T$ such that $(s_i, s_{i+1}) \models \rho$. Likewise, an *infinite computation* of the program P is an infinite sequence $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots, \langle \ell_k, s_k \rangle \ldots \in (L \times v.X)^\omega$, where ℓ_0 is the initial location and for each $i \geq 0$, there is a transition $(\ell_i, \rho, \ell_{i+1}) \in T$ such that $(s_i, s_{i+1}) \models \rho$. A computation is either a finite computation or an infinite computation. A state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some computation. A location ℓ is reachable if there exists some state s such that $\langle \ell, s \rangle$ appears in some computation. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions, where ℓ_0 is the initial location.

We define two useful operations on states. For a state s and a constraint ρ over $X \cup X'$, we define the set of *successor states* $Post(s, \rho) = \{s' \mid (s, s') \models \rho\}$. Similarly for a state s' and constraint ρ, we define the set of *predecessor states* $Pre(s', \rho) = \{s \mid (s, s') \models \rho\}$. The operations *Post* and *Pre* are extended to sets of states in the obvious way: $Post(S, \rho) = \bigcup_{s \in S} Post(s, \rho)$ and $Pre(S, \rho) = \bigcup_{s \in S} Pre(s, \rho)$. The *Post* and *Pre* operations are also called the *strongest postcondition* and *weakest liberal precondition* operations respectively [Dijkstra 1976].

1.2 Properties

Let P be a simple program, and let $\mathcal{E} \in L$ be a special *error* location. The program is *safe* w.r.t. the error location $\mathcal{E}$ if the location $\mathcal{E}$ does not appear in any computation. An *error trace* is a computation ending in the location $\mathcal{E}$.

The safety verification problem takes as input a program P and an error location $\mathcal{E} \in L$, and returns "safe" if P is safe w.r.t. $\mathcal{E}$, and "unsafe" if $\mathcal{E}$ is reachable.

While we formulate the safety verification problem as a check for reachability of a particular location, it is known that checking any safety property (expressed, *e.g.,* in a temporal logic) can be reduced to the above reachability question. An alternate and common way to specify safety conditions for programs is through *assertions*. The programmer explicitly puts in a predicate p over program variables (called the assertion) at a program location, with the intent that for every execution of the program that reaches the location, the program state satisfies the assertion. Our simple formulation subsumes *assertion checking* in a simple way: each assertion is replaced by a conditional on the predicate p, the program continues execution if the predicate p holds, and enters an error location otherwise.

In the following, we use the following terminology. An algorithm for the safety verification problem is *sound* if for every program P and error location $\mathcal{E}$ of P, if the algorithm returns "safe" then P is safe w.r.t. $\mathcal{E}$. It is *complete* if for every program P and error location $\mathcal{E}$ of P, if P is safe w.r.t. $\mathcal{E}$, then the algorithm returns "safe".

1.3 Organization

The rest of the survey is organized as follows. We first describe two main ways of representing state: enumerative (Sections 2 and 3) and symbolic (Section 4). We then describe abstraction techniques (Section 5), which reduce the state space at the expense of precision, and automatic techniques to make abstract analyses more precise (Section 6). While we describe each facet in isolation, in practice, several notions can be combined within the same tool: *e.g.,* the program state can be represented partly in enumerated form, and partly symbolically, and combined with (several different) abstractions.

The next few sections describe extensions to the basic approach: dealing with (potentially recursive) functions (Section 7), dealing with heaps with dynamic allocation and heap abstractions (Section 8), and extending the techniques to reason about liveness properties (Section 9). We then make the connection between model checking techniques and related dynamic (testing) and static (type systems) techniques in software quality (Section 10). We conclude with the limitations of current tools and some future challenges.

2. CONCRETE ENUMERATIVE MODEL CHECKING

Let $\mathsf{imp}_{\mathsf{fin}}$ be the class of simple programs where each variable ranges over a finite domain. In this case, the safety verification problem can be solved by explicitly constructing the (finite) set of reachable states and checking that $\mathcal{E}$ is not reachable.

Figure 1 shows Algorithm Reachability, a procedure that computes the set of reachable states of a program in $\mathsf{imp}_{\mathsf{fin}}$ by performing graph search. The algorithm maintains a set *reach* of *reachable states* and a set *worklist* of *frontier states* that are found to be reachable but whose successors may not have been explored. Initially, the set *reach* is empty, and the frontier *worklist* contains all the *initial states*. The main *reachability loop* of the algorithm explores the states of the frontier one at a time. If the state has not been visited before, the successors of the state are added to the frontier, otherwise, the successors are not added. The process is repeated until all reachable states have been explored, which happens when the frontier *worklist* becomes empty. At this point, *reach* contains exactly the set of reachable states. The loop terminates for all $\mathsf{imp}_{\mathsf{fin}}$, in fact, it terminates for all programs for which *reach* is finite.

While we have implemented the check for reachability of $\mathcal{E}$ at the end of the reachability loop, it can be performed whenever a new state is picked from the frontier. Also, Algorithm 1 can be easily modified to produce an error trace in case $\mathcal{E}$ is reachable,

The generic schema of Figure 1 can be instantiated with different data structures for maintaining the set of reachable states and the set of frontier states, and with algorithms for implementing the order in which states are chosen from the frontier set.

For example, maintaining the frontier as a stack (and always choosing the next state by popping the stack) ensures depth-first traversal of the graph, while maintaining the frontier as a queue ensures breadth-first traversal. For efficiency of checking membership, the set *reach* is usually implemented as a hashtable [Holzmann 1997; Dill 1996]. In addition, instead of generating all states and tran-

Algorithm: Enumerative Reachability
Input simple program $P = (X, L, T, \ell_0)$, error location $\mathcal{E} \in L$
Output SAFE if P is safe w.r.t. $\mathcal{E}$, UNSAFE otherwise

```
def Reachability(P, E):
    reach = ∅
    worklist = {(ℓ0, s) | s ∈ v.X}
    while worklist ≠ ∅ do:
        pick and remove(ℓ, s) from worklist
        if (ℓ, s) ∉ reach:
            add (ℓ, s) to reach
            foreach (ℓ, ρ, ℓ') in T do:
                add {(ℓ', s') | s' ∈ Post(s, ρ)} to worklist
    if exists (E, s) ∈ reach:
        return UNSAFE
    else:
        return SAFE
```

Fig. 1. Enumerative Reachability

sitions up front, reachability search algorithms usually construct the state space *on-the-fly*, based on currently reached states and the program. This exploits the observation that the reachable state space of the program can be much smaller than the state space.

While we focus on a *forward* algorithm, based on the *Post* operator, a dual *backward* algorithm based on the *Pre* operator is also possible. This algorithm starts at the $\mathcal{E}$ location and searches backward to see if some initial state can be reached.

Enumerative model checking of finite state concurrent programs has been implemented in several tools, most notably SPIN[Holzmann 1997] and MURPHI [Dill 1996]. Both tools have had significant impact, especially in the protocol verification domain.

The state space of a program can be exponentially larger than the description of the program. This problem, known as *state explosion*, is one of the biggest stumbling blocks to the practical application of model checking. Controlling state explosion has therefore been a major direction of research. Broadly, the research takes two directions.

First, *reduction-based* techniques compute equivalence relations on the program behaviors, and explore one (or a few) candidates from each equivalence class. A meta-theorem asserts that the reduced exploration is complete, that is, for any bug in the original system, there is a bug in the reduced one. Primary reduction-based techniques consist of *partial-order reduction* [Godefroid 1996], *symmetry reduction* [Emerson and Sistla 1996; Ip and Dill 1996; Sistla et al. 2000] or minimizations based on behavioral equivalences such as simulation or bisimulation [Bouajjani et al. 1990; Loiseauxc et al. 1995; Bustan and Grumberg 2003]. Partial order reductions exploit the independence between parallel threads of execution on unrelated parts of the state, choosing to explore one candidate interleaving rather than all of them. Symmetry reduction determines symmetries in the program, and explores one element from each symmetry class.

Second, *compositional* techniques [Chandy and Misra 1988;

Abadi and Lamport 1990; Alur and Henzinger 1999; Alur et al. 1998] reduce the safety verification problem on the original program to related problems on smaller parts of the program, such that the results on the parts can be combined to check the safety of the original program.

In a different direction, using the observation that model checking is often more useful as a *falsification* or bug-finding aid, one gives up completeness of the search. This is often done by limiting resources available to the model checker (run for a specified amount of time or memory), or by bounding the set of behaviors of the program to be explored (*e.g.,* by bounding the depth of the search, or the number of context-switches). *Bitstate hashing* is a popular technique in which the hash of each reachable state is stored, rather than the state itself. The choice of the range of the hash function is determined by the available memory. Bitstate hashing is unsound, as two distinct reached states can hash to the same value (a hash *collision*). In order to obtain nicer guarantees on the probability of collision, each state is hashed using several (in practice, two or three) independent hash functions. When the search space is so big that even with bitstate hashing one can only explore a small portion of the state space, it is possible to store the states on disk rather than on RAM [Stern and Dill 1998; Korf 1985]. This is a time-for-space tradeoff: while accessing states take much longer on disk, the disk allows storing a much larger state space.

The search heuristic also has a profound influence on the performance of Algorithm 1 on practical problems, where the objective is to find bugs quickly. One direction of research applies *guided* or *directed search* heuristics inspired by search heuristics from the artificial intelligence literature, such as iterative deepening [Korf 1985], best-first search [Russell and Norvig 2003], A* search [Hart et al. 1968; Russell and Norvig 2003], or perimeter search [Dillenburg and Nelson 1994]. These techniques were imported into Murphi [Yang and Dill 1998] and Spin [Edelkamp et al. 2004; Lluch-Lafuente 2003], and there have been several extensions (and combinations with orthogonal techniques) since [Fraser et al. 2000; Edelkamp et al. 2004]. In the presence of multi-core machines, one strategy is to allocate fixed budgets and run different search strategies, including random or biased random searches, in parallel [Holzmann et al. 2008].

3. MODEL CHECKING BY SYSTEMATIC EXECUTION EXPLORATION

The execution-based model checking approach, pioneered by VERISOFT [Godefroid 1997], trades complete coverage (*i.e.,* formal verification) for the ability to find bugs in arbitrary software.

In the execution based approach, the non-determinism in a concurrent program is factored into two sources: *inputs* from the environment, and *scheduling choices* made by the scheduler. That is, each sequence of user inputs and schedule choices uniquely determines the outcome of an execution, and the set of all behaviors can be explored by analyzing the behavior of the process under all possible inputs and schedules.

In contrast to techniques that exhaustively explore the state space using graph algorithms, systematic execution exploration proceeds by systematically iterating over the space of possible schedules and simply *executing* the process under each

schedule. This most appealing benefit of this approach is that it sidesteps the need to be able to formally represent the semantics of the programming language and machine instructions as a transition relation. In essence, the transitions correspond directly to the manner in which the machine state is altered by the execution of instructions between scheduler invocations. Moreover, when a counterexample is found, the model checker can generate a concrete execution demonstrating how the system violates the property. This counterexample is typically detailed enough to be replayed inside a debugger (*e.g.,* GDB), thereby helping the user pinpoint the cause of error.

Execution-based model checkers are typically used as a "test amplification" mechanism. The user provides a test harness corresponding to a workload under which the program is run. The usual operating system scheduler would execute the workload under a fixed schedule, thereby missing most possible behaviors. However, the execution-based model checker's scheduler is able to systematically explore the possible executions of the same workload under different schedules, *e.g.,* exploring what happens under different interleavings of shared operations like message sends, receives *etc.*, and is able to find various safety errors like assertion failures, deadlocks and divergences, that are only manifested under corner-case schedules.

However, there are two technical challenges that must be addressed to make execution-based model checking feasible. These challenges stem from the fact that in this approach the "state" comprises the entire machine state – all the registers, the heap, the stack, state of network resources *etc.*. First, the size of the machine state makes it infeasible to store the set of visited states. Thus, how can one systematically explore the space of executions *without* storing the visited states? Second, the space of possible schedules grows exponentially in the length of the schedule. However, many of these schedules can lead to states that are either identical, or identical with respect to the properties being analyzed (*i.e.,* differing only in the values of some irrelevant variables like performance counters). Given that one can only run the model checker for a finite amount of time, how can one *bias* the analysis away from exploring redundant executions and towards executions that are more likely to expose bugs? There is the usual time-space-soundness tradeoff here: storing entire states ensure paths are not re-executed but require too much space; storing no states ("stateless search" described below) spends time re-executing similar paths and can diverge if there are loops; and storing hashes of states can miss executions owing to hash collisions.

3.1 Stateless Search

The key innovation in VERISOFT was to introduce the idea of stateless search, wherein the model checker could explore different executions without actually storing the set of visited states. Since all non-determinism is captured within the schedules, whenever the process needs to make a non-deterministic choice, *e.g.,* a choice based on a random number, a user input, the choice of thread to execute or network latency, it queries the scheduler for a number from a finite range. Each execution is characterized by the *schedule*, that is, the sequence of numbers returned by the scheduler to the process. Thus, to iterate over different executions, the model checker needs only to iterate over the space of possible schedules and re-execute the process using the schedule.

```
Algorithm: Stateless Search
Input Program P, Depth D.
Output NoErrorsFound in schedules of size ≤ D, Unsafe otherwise.

def StatelessSearch(P, D):
  depth = 0
  while depth ≤ D do:
    increment depth
    foreach schedule in Sequences(depth) do:
      Reset(sys)
      if Execute(P, schedule) reaches an error :
        return Unsafe
  return NoErrorsFound
```

Fig. 2. StatelessSearch

While conceptually the schedule does not distinguish user inputs and thread schedule choices, even for a small number of 32-bit user inputs, the number of schedules can become astronomical. VERISOFT (and many similar tools described below) explicitly model the non-determinism arising from scheduling choices, keeping user inputs fixed. Tools based on symbolic execution (described in Section 4) for sequential program focus on the other hand on exploring the space of user inputs, using symbolic representations to avoid the explosion in schedules. Tools such as JAVAPATHFINDER combine both approaches, exploring schedule choices explicitly while tracking user inputs symbolically.

Algorithm StatelessSearch in Figure 2 shows this algorithm. The algorithm takes as input a system and a depth increment, and performs a *bounded-depth depth first search* of the space of executions. The size of an execution is measured by the number of calls made to the scheduler during the execution.

Recall that each execution of size *depth* is characterized by the sequence of values that the scheduler returns along that execution. Suppose that at each point, the scheduler returns an integer between 0 and k. *Sequences*(*depth*) is an iterator over the sequences of $1 \ldots k$ of size *depth*. That is, it returns, in order, the sequences,

$$0^{depth}, 0^{depth-1}1, \ldots, 0^{depth-1}k, 0^{depth-2}10, \ldots, k^{depth}$$

For each sequence or schedule, we analyze the behavior of the system under the schedule by resetting the system to its initial state and executing the system under the schedule. The order in which sequences of a given depth *depth* are generated ensures that algorithm explores executions in a depth-first manner. Once all the executions of size *depth* have been explored, the *depth* is incremented and the process is repeated until the execution returs UNSAFE meaning that an execution along which some safety property has been violated has been found.

The key properties of the above algorithm are that it makes very modest demands on the representation of the process being analyzed. We need only to be able to: (1) *reset* it, or restore it to a unique starting state, and, (2) *execute* it, under a given schedule. Notice that to explore executions of larger depths, the algorithm simply re-executes the system from the initial state. This requires that the execution be deterministic under a given schedule. That is, two executions using the same schedule leave the system in identical states. In practice, the most onerous burden

upon the user is to ensure that *every* source of non-determinism is replaced with a call to the scheduler.

3.2 Effective Exploration

In execution-based model checking, the number of sequences, and hence possible executions, of size n is k^n where k is the number of possible values returned by the scheduler at each step. Next, we describe several execution-based model checkers, and the strategies they have adopted to effectively explore the executions in order to find bugs.

Verisoft. The idea of execution-based stateless software model checking was pioneered by VERISOFT, which views the entire system as the composition of several Unix *processes* that communicate by means of message queues, semaphores and shared variables that are *visible* to the VERISOFT scheduler. The scheduler traps calls made to access the shared resources, and by choosing which process will execute at each trap point, the scheduler is able to exhaustively explore all possible interleavings of the processes' executions. As the shared operations are exposed to the scheduler, it can perform *partial-order reduction* [Godefroid 1996] to avoid reanalyzing equivalent executions. VERISOFT has been used to found several complex errors in concurrent phone-switching software [Chandra et al. 2002].

JavaPathFinder. JAVAPATHFINDER is an execution-based model checker for Java programs that uses the *Java Virtual Machine* rather than the OS scheduler, to analyze different schedules [Visser et al. 2003; Havelund and Pressburger 2000]. This language-based approach restricts the classes of software that can be analyzed, but provides many significant advantages. First, the use of the JVM makes it possible to store the visited states, which allows the model checker to use many of the standard reduction-based approaches (*e.g.,* symmetry, partial-order, abstraction) to combating state-explosion. Second, as the visited states can be stored, the model checker can utilize various search-order heuristics without being limited by the requirements of stateless search. Third, one can use techniques like symbolic execution and abstraction to compute inputs that force the system into states that are different from those previously visited thereby obtaining a high level of coverage. JAVAPATHFINDER has been used to successfully find subtle errors in several complex Java components developed inside NASA [Brat et al. 2004; Penix et al. 2005], and is available as a highly extensible open-source tool.

Cmc. CMC is an execution based model checker for C programs that explores different executions by controlling schedules at the level of the OS scheduler. CMC stores a hash of each visited state. In order to identify two different states which differ only in irrelevant details like the particular addresses in which heap-allocated structures are located, CMC *canonicizes* the states before hashing to avoid reexploring states that are similar to those previously visited. CMC has been used to find errors in implementations of network protocols [Musuvathi and Engler 2004] and file systems [Yang et al. 2004].

4. SYMBOLIC MODEL CHECKING

Symbolic algorithms manipulate (representations of) *sets* of states, rather than individual states. For example, the constraint $1 \leq x \leq 10 \wedge 1 \leq y \leq 8$ represents the

set of all states over $\{x, y\}$ satisfying the constraint. Thus, the constraint *implicitly* represents the list of 80 states that would be enumerated in enumerative model checking. Symbolic representations of sets of states can be much more succinct than the corresponding enumeration, and can represent infinite sets of states as well. The *symbolic representation* of regions is the crucial component of symbolic algorithms: in addition to providing an implicit representation of sets of states, the representation must allow performing operations on sets of states directly on the representation. For example, given a program with (symbolic) initial region σ_I, the set of reachable states are given by $\bigcup_{i \geq 0} Post^i(\sigma_I)$. This suggests that a symbolic representation of regions must allow at least the *Post* and the $\cup$ operations on symbolic regions, and a way to check inclusion between regions (to check for convergence).

4.1 The Region Data Structure

Let P be a simple program. For symbolic model checking algorithms, we introduce the abstract data type of *symbolic representations* for states of P. The abstract data type defines a set of symbolic regions symreg, an *extension function* $[\![\cdot]\!] : \mathsf{symreg} \rightarrow 2^{v.X}$ mapping each symbolic region to a set of states, and the following constants and operations on regions:

(1) The constant $\bot \in \mathsf{symreg}$ representing the empty set of states, $[\![\bot]\!] = \emptyset$, and the constant $\top \in \mathsf{symreg}$ representing the set of all states, $[\![\top]\!] = v.X$.

(2) The operation $\cup : \mathsf{symreg} \times \mathsf{symreg} \rightarrow \mathsf{symreg}$ that computes the union of two regions, that is, for every $r, r' \in \mathsf{symreg}$ we have $[\![r \cup r']\!] = [\![r]\!] \cup [\![r']\!]$.

(3) The operation $\cap : \mathsf{symreg} \times \mathsf{symreg} \rightarrow \mathsf{symreg}$ that computes the intersection of two regions, that is, for every $r, r' \in \mathsf{symreg}$ we have $[\![r \cup r']\!] = [\![r]\!] \cap [\![r']\!]$.

(4) The operation $=: \mathsf{symreg} \times \mathsf{symreg} \rightarrow \mathbf{bool}$ such that $r = r'$ returns *true* iff r and r' denote the same set of states, *i.e.,* $[\![r]\!] = [\![r']\!]$.

(5) The operation $\subseteq: \mathsf{symreg} \times \mathsf{symreg} \rightarrow \mathbf{bool}$ such that $r \subseteq r'$ returns *true* iff r denotes a region contained in r', *i.e.,* $[\![r]\!] \subseteq [\![r']\!]$.

(6) The operation $Post : \mathsf{symreg} \times T \rightarrow \mathsf{symreg}$ that takes a symbolic region r and a transition relation τ and returns a symbolic region denoting the set $Post([\![r]\!], \tau)$.

(7) The operation $Pre : \mathsf{symreg} \times T \rightarrow \mathsf{symreg}$ that takes a symbolic region r and a transition relation τ and returns a symbolic region denoting the set $Pre([\![r]\!], \tau)$.

In what follows, we shall assume that each operation above is effectively computable for an implementation of symreg.

Figure 3 shows Algorithm Reachability, a symbolic implementation of reachability. The basic search procedure is the same as Algorithm Reachability (Figure 1), but we now manipulate symbolic regions rather than individual states. Note that we maintain the program locations enumeratively, alternately, we can construct a fully symbolic version where the program locations are maintained symbolically as well. The algorithm maintains a map *reach* from locations $\ell \in L$ to sets of states reachable when the control location is ℓ. The frontier regions are maintained in *worklist* as before, and the main loop explores regions at the frontier and adds new regions to *reach*. (The notation $reach[\ell \mapsto r]$ denotes a function which maps ℓ to r but agrees with *reach* on all other locations.)

Algorithm: Symbolic Reachability
Input simple program $P = (X, L, \ell_0, T)$, error location $\mathcal{E} \in L$
Output SAFE if P is safe w.r.t. $\mathcal{E}$, UNSAFE otherwise

```
def Reachability(P, E):
    reach = λℓ.⊥
    worklist = (ℓ0, ⊤)
    while worklist ≠ ∅ do:
        pick and remove(ℓ, r) from worklist
        if r ⋢ reach(ℓ):
            reach = reach[ℓ ↦ r ∪ reach(ℓ)]
            foreach (ℓ, ρ, ℓ') in T do:
                add (ℓ', Post(r, ρ)) to worklist
    if reach(E) ≠ ⊥:
        return UNSAFE
    else:
    return SAFE
```

Fig. 3. Symbolic Reachability

Notice that the algorithm does not make any assumptions of finiteness: as long as the symbolic regions can represent infinite sets of states, and the symbolic operations are effective, the algorithm can be implemented.

The power of symbolic techniques comes from tremendous advances in the performance of constraint solvers that underlie effective symbolic representations, both for propositional logic (satisfiability solvers [Silva and Sakallah 1996; Moskewicz et al. 2001; Een and Sorensson 2003] as well as binary decision diagrams [Bryant 1986; Somenzi 1998]) and more recently for combinations of first order theories [Dutertre and Moura ; Bruttomesso et al. 2008; de Moura and Bjørner 2008].

4.2 Example: Propositional Logic

For finite domain programs, Boolean formulas provide a symbolic representation. The encoding is easiest understood when each variable in the program is Boolean (finite domain variables can be encoded using several Boolean variables). Suppose the program has n Boolean variables. Then, a state is a vector of n bits, and each set of states can be represented by its characteristic formula that maps an n bit vector to *true* if it is in the set and to *false* otherwise.

In this representation, the regions $\bot$ and $\top$ are then the formulas *false* and *true*, respectively, Boolean operations on regions are logical operations on formulas, and equality and subset checks reduce to Boolean equivalence and implication checks. Finally, *Post* and *Pre* can be computed by Boolean conjunction followed by existential quantifier elimination (and renaming) as follows. Given a transition relation $\tau(X, X')$ represented as a BDD over $2n$ variables (the unprimed and primed variables), and a BDD $b(X)$ over the current set of variables, $Post(b(X), \tau(X, X'))$ is given by the BDD

$$\mathsf{Rename}(\exists X.b(X) \wedge \tau(X, X'), X', X)$$

where the existential quantifier $\exists X$ existentially quantifies each variable in X, and $\mathsf{Rename}(\phi(X'), X', X)$ renames each primed variable $x' \in X'$ appearing in the BDD $\phi(X')$ by the corresponding unprimed variable $x \in X$. Similarly,

$Pre(b(X), \tau(X, X'))$ is given by the BDD

$$\exists X'.\mathsf{Rename}(b(X), X, X') \wedge \tau(X, X')$$

Unfortunately, it is often difficult to work directly with Boolean formulas. (Reduced ordered) binary decision diagrams (BDDs) [Bryant 1986] can be used instead as an efficient implemention of Boolean formulas. BDDs are compact and canonical representations of Boolean functions.

For a program with n state bits, regions are represented as BDDs over n variables with some ordering on the variables. As before, the empty region is the BDD for *false*, representing the empty set, and the top region $\top$ is the BDD for *true*, representing all states. Union and intersection are Boolean disjunction and conjunction, respectively, which can be computed directly on BDDs [Bryant 1986]. Equality is Boolean equivalence, but reduces to equality on BDDs since they are canonical representations. Checking containment reduces to equality. The computation of *Post* and *Pre* require existential quantification and renaming (in addition to Boolean operations), which can again be implemented directly on BDDs.

BDDs are the primary representation in symbolic model checkers such as SMV, and have been instrumental in scaling hardware model checkers to extremely large state spaces [Burch et al. 1992; McMillan 1993]. Each Boolean operation and existential quantification of a single variable can be quadratic, and the size of the BDD can be exponential in the number of variables in the worst case. Moreover, the size of the BDD is sensitive to the variable ordering, and many functions do not have a succinct BDD representation [Bryant 1986]. This is the symbolic analogue of the state explosion problem, and has been a major research direction in model checking.

Notice that Algorithm 3 terminates for $\mathsf{imp}_{\mathsf{fin}}$ where symreg is implemented using BDDs, since each BDD operation is effective, and each iteration of the loop finds at least one new state.

4.3 Example: First Order Logic with Interpreted Theories

In case program variables range over infinite domains, such as integers, a more expressive symbolic representation is provided by first order logic formulas, often over interpreted theories [Hoare 1969; Nelson 1981]. A symbolic region is represented by a first order formula whose free variables range over the program variables. For a formula φ, we write $[\![\varphi]\!]$ for the set of states $\{s \mid s \models \varphi\}$. With abuse of notation, we often identify a formula φ with the set of states $[\![\varphi]\!]$, and omit $[\![\cdot]\!]$ when clear from the context. The empty region $\bot$ is represented by the logical formula *false*, the region $\top$ by *true*. Union and intersection are implemented by disjunction and conjunction in the logic, and equality and containment by equivalence checking and implication, respectively.

Finally, the *Pre* and *Post* operations can again be implemented using existential quantification, using essentially the same formulations as for the Boolean case above, but where the formulas can now be over the more general logic.

When using the full power of first order logic, the equivalence and implication checks are not effective. So in practice, one chooses formulas over a *decidable* theory, such as quantifier-free formulas over a combination of the theory of equality with uninterpreted functions and the theory of rationals [Nelson 1981]. While this

makes the individual operations in Algorithm 3 effective, it could still be that the algorithm runs forever. Consider for example a loop

$$\mathtt{i} := 0; \mathbf{while}\ (\mathtt{i} \geq 0)\ \{\mathtt{i} := \mathtt{i} + 1;\}\mathcal{E}:;$$

Clearly, the location $\mathcal{E}$ is not reachable. However, a symbolic representation based on first order formulas could run forever, finding the set of reachable states

$$\mathtt{i} = 0 \vee \mathtt{i} = 1 \vee \ldots \vee \mathtt{i} = k$$

at the k-th iteration of the while loop, approximating the "fixed point" $\mathtt{i} \geq 0$ closer and closer.

4.4 Bounded Model Checking

As in enumerative model checking, one can trade-off soundness for effective bug finding in symbolic model checking. One popular approach, called *bounded model checking* [Biere et al. 1999], unrolls the control flow graph for a fixed number of steps, and checks if the error location can be reached within this number of steps. Precisely, given program P, error location $\mathcal{E}$, and $k \in \mathbb{N}$, one constructs a constraint which is satisfiable iff the error location $\mathcal{E}$ is reachable within k steps. Satisfiability of this constraint is checked by a constraint solver. The technique is also called *symbolic execution*, especially when applied to a particular path in the control flow graph, since the program is executed on symbolic as opposed to concrete inputs.

Tools for bounded model checking of software implementations come in two flavors. The first, such as CBMC [Kroening et al. 2003] or Saturn [Xie and Aiken 2005], generate constraints in propositional logic and use Boolean satisfiability solvers to discharge the constraints. The reduction to propositional satisfiability captures the semantics of fixed-width program datatypes precisely. Thus, one can find subtle bugs arising from mismatches between the algorithm and low-level fixed-width machine semantics, such as arithmetic overflows.

CBMC [Kroening et al. 2003] and Saturn [Xie and Aiken 2005] are tools implementing this idea for C programs. Both have been fairly successful in analyzing large pieces of software, including analyzing C models of processors and large parts of the Linux kernel. Saturn improves upon the basic bounded model checking algorithm by computing and memoizing relations between inputs and outputs ("summaries") for procedures bottom-up in the call graph. This makes bounded model checking scale to large programs. Bounded model generation ideas have also been used to check consistency of abstract specifications, most notably in the Alloy analyzer [Jackson 2006].

The second class of tools generates constraints in an appropriate first order theory (in practice, the combination theory of equality with uninterpreted functions, linear arithmetic, arrays, and some domain-specific theories) and use decision procedures for such theories [de Moura et al. 2002; Ivancic et al. 2008; Armando et al. 2006]. The basic algorithm is identical to SAT-based bounded model checking, but the constraints are interpreted over more expressive theories.

4.5 Invariants and Invariant Synthesis

The set of reachable states of a program can be viewed as the least fixpoint of the functional which maps a set of states r to the set of states reachable from r in

at most one step. This least fixpoint is computed iteratively and symbolically by Algorithm Reachability. For infinite-state systems, the computation of the fixpoint is not guaranteed to terminate in a finite number of steps. One way of ensuring termination is by explicitly providing regions which represent fixpoints (not necessarily the least fixpoint) of the functional, and *check* that these regions are indeed fixpoints, rather than compute the least fixpoint from scratch. These regions are called *invariants*.

An *invariant* of P at a location $\ell \in L$ is a set of states containing the states reachable at ℓ. An *invariant map* is a function η from L to formulas over program variables from X such that the following conditions hold:

Initiation: For the initial location ℓ_0, we have $\eta.\ell_0 = \mathit{true}$.
Inductiveness: For each $\ell, \ell' \in L$ such that $(\ell, \rho, \ell') \in T$, the formula $\eta.\ell \wedge \rho$ implies $(\eta.\ell')'$. Here, $(\eta.\ell')'$ is the formula obtained by substituting variables from X' for the variables from X in $\eta.\ell'$.
Safety: For the error location $\mathcal{E}$, we have $\eta.\mathcal{E} = \mathit{false}$.

With these properties, it can be proved by induction that at every location $\ell \in L$, we have that $\{s \mid (\ell, s) \text{ is reachable in } P\} \subseteq [\![\eta.\ell]\!]$, and so the method is sound. In the above example, a possible invariant map associates the invariant $\mathtt{i} \geq 0$ with the head of the while loop.

The presence of invariants reduces the problem of iteratively computing the set of fixed points to checking a finite number of obligations, which is possible if the symbolic representation is effective. In fact, relative completeness results [Dijkstra 1976] ensure that relative to deciding the implications, the method is complete for safety verification. Traditional program verification has assumed that the invariant map is provided by the programmer, and several existing tools (ESC-Modula [Leino and Nelson 1998], ESC-Java [Flanagan et al. 2002]) check the conditions for invariants, given an invariant map, using decision procedures. (For ease of exposition, we have assumed that an invariant map assigns an invariant to each program location, but it is sufficient to define invariants only over a program *cutset*, *i.e.,* a set of program locations such that every syntactic cycle in the CFG passes through some location in the cutset. Thus, the tools only require *loop invariants* from the programmer.)

In practice, the requirement that the programmer provide invariants has not found widespread acceptance, so the emphasis has shifted to algorithmic techniques that can synthesize invariant maps automatically, with minimal support from the programmer. Clearly, the set of reachable states at each location constitutes an invariant map. However, there are other invariant maps as well, each of which contains the set of reachable states. This "relaxation" enables the use of *abstraction techniques* for invariant synthesis that we look at in the next section.

We close this section with a constraint-based algorithm for invariant synthesis. The method starts with a *template invariant map*, *i.e.,* a parameterized expression representing the invariant map, and encodes the three conditions (initiation, inductiveness, safety) on the templates to get a set of constraints on the parameters of the template [Giesl and Kapur 2001; Sankaranarayanan et al. 2005]. A solution to these constraints provides an assignment to the parameters in the template and constitute an invariant map. For templates in linear rational arithmetic, the con-

straint system can be encoded as a set of *non-linear* arithmetic constraints on the template parameters [Sankaranarayanan et al. 2005], and decision procedures for real arithmetic can be used to construct invariant maps. Invariant synthesis for templates over ore expressive theories, such as combinations of linear arithmetic and equality with uninterpreted functions, can be reduced to non-linear arithmetic constraints [Beyer et al. 2007].

While attractive, the technique is limited by two facts. First, the programmer has to guess the "right" template. Second, the scalability is limited by the performance of constraint solvers for non-linear arithmetic. Thus, so far constraint-based invariant synthesis has been applied only to small programs.

5. MODEL CHECKING AND ABSTRACTION

For infinite state programs, symbolic reachability analysis may not terminate, or take an inordinate amount of time or memory to terminate. Abstract model checking trades off precision of the analysis for efficiency. In abstract model checking, reachability analysis is performed on an *abstract domain* which captures some, but not necessarily all, the information about an execution, using an abstract semantics of the program [Cousot and Cousot 1977]. A proper choice of the abstract domain and the abstract semantics ensures that the analysis is sound (*i.e.,* proving the safety property in the abstract semantics implies the safety property in the original, *concrete*, semantics) and efficient.

Historically, the two fields of model checking and abstract interpretation have evolved in parallel, with model checking emphasizing precision and abstract interpretation (in the form of program analysis for compilation) emphasizing efficiency. It has been long known that theoretically each approach can simulate the other [Cousot and Cousot 2000; Schmidt 1998], and more recently, the traditional divide between the two disciplines have become blurred [Fischer et al. 2005]. In what follows, we focus on abstract *reachability* analysis, but the techniques generalize to model checking more expressive temporal properties, for example, those expressed in the μ-calculus [Clarke et al. 1992].

5.1 Abstract Reachability Analysis

We focus on the pragmatics of an abstract reachability analysis algorithm and omit the theory of abstract interpretation (see, *e.g.,* [Cousot and Cousot 1977; Nielson et al. 1999]).

An *abstract domain* $(\mathcal{L}, [\![\cdot]\!], Pre^{\#}, Post^{\#})$ for a program P consists of a complete lattice $\mathcal{L} = (L, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ of *abstract elements*, a *concretization function* $[\![\cdot]\!] : L \to 2^{v.S}$ mapping lattice elements to sets of states, and two monotone total functions $Pre^{\#} : L \times T \to L$ and $Post^{\#} : L \times T \to L$ such that the following conditions hold:

(1) $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = v.S$;
(2) for all elements $l, l' \in L$, we have $[\![l \sqcup l']\!] \supseteq [\![l]\!] \cup [\![l']\!]$ and $[\![l \sqcap l']\!] \supseteq [\![l]\!] \cap [\![l']\!]$; and
(3) for all $l \in L$ and transition ρ, we have $[\![Post^{\#}(l, \rho)]\!] \supseteq Post([\![l]\!], \rho)$ and $[\![Pre^{\#}(l, \rho)]\!] \supseteq Pre([\![l]\!], \rho)$.

A lattice element $l \in L$ represents an *abstract* view of a set of program states $[\![l]\!]$. Note that $[\![\cdot]\!]$ is not required to be onto: not all sets of program states need to have

an abstract representation. A strictly ascending chain is a sequence $l_0 \sqsubset l_1 \sqsubset l_2 \ldots$. The *height* of a lattice is the cardinality of the largest strictly ascending chain of elements in L.

By replacing symbolic regions by abstract domains in Algorithm 3, we get an abstract reachability algorithm. From the properties of an abstract domain, the abstractly reachable set $reach^{\#}$ so computed has the property $reach \subseteq [\![reach^{\#}]\!]$, and so if the abstract reachability algorithm returns "safe" then the program is indeed safe w.r.t. the error location. Unfortunately, the converse is not true in general: the abstract reachability could return "unsafe" if it loses too much precision in the abstraction process, even though the program is safe.

Consider the example program shown in Figure 5. This program is a simplified version of a function from a network device driver [Ball and Rajamani 2002b]. Intuitively, the variable `LOCK` represents a global lock; when the lock has been acquired, the value of `LOCK` is 1, a and when the lock is released, the value of `LOCK` is 0. We would like to verify that at the end of the do-loop, the lock is acquired (i.e., $\texttt{LOCK} = 1$). In the code, this assertion is specified by a check that the lock is acquired (on line 5) and a call to `error` if the check fails. If the abstraction consists of the relations $\texttt{LOCK} = 0$, $\texttt{LOCK} = 1$, $\texttt{old} = \texttt{new}$, and $\texttt{old} \neq \texttt{new}$, the abstract reachability analysis can prove this property. The abstraction of the set of reachable set of states at line 4 is

$$(\texttt{LOCK} = 1 \wedge \texttt{new} = \texttt{old}) \vee (\texttt{LOCK} = 0 \wedge \texttt{new} \neq \texttt{old})$$

which captures the intuition that at line 4, either the lock is acquired and `new` is equal to `old`, or the lock is not acquired and the value of `new` is different from `old` (in fact, $\texttt{new} = \texttt{old} + 1$).

On the other hand, if the program is analyzed using only the predicates $\texttt{LOCK} = 0$ and $\texttt{LOCK} = 1$, the abstract reachability analysis does not track the relationship between `new` and `old`, and hence cannot prove the property.

Notice that if the abstract domain has finite height, the abstract reachability algorithm is guaranteed to terminate. Even if the abstract domain has infinite height, the abstract reachability algorithm is still applicable, but usually augmented with *acceleration* or *widening* techniques to ensure termination in a finite number of steps, at the cost of reduced precision [Cousot and Halbwachs 1978] (see discussion on polyhedral domains below).

5.2 Example: Polyhedral Domains

In the *polyhedral abstract domain*, the abstract elements are polyhedral sets over an n-dimensional space of program variables ordered by set inclusion. Intersection is implemented as polyhedron intersection, and union as convex hull. The $Pre^{\#}$ and $Post^{\#}$ operations are implemented as intersections and projections of polyhedra for transition relations ρ that are linear relations on $X \cup X'$, and approximated for others.

The polyhedral domain has been successfully used to check for array bounds checks [Cousot and Halbwachs 1978], and efficient implementations are available (*e.g.,* [Bagnara et al. 2008]). The domain is not finite height, so the abstract reachability algorithm may not terminate after a finite number of steps. To ensure termination, the reachability analysis uses a *widening* operator

[Cousot and Halbwachs 1978].

Faster, but less expressive, abstract domains that can represent a subclass of polyhedra, such as intervals [Cousot and Cousot 1976], difference-bound matrices [Larsen et al. 1997], or octagons [Miné 2006] have been used as well.

5.3 Example: Predicate Abstraction

The *predicate abstraction* domain [Agerwala and Misra 1978; Graf and Saïdi 1997; Das et al. 1999] is parameterized by a fixed finite set Π of first order formulas with free variables from the program variables, and consists of the lattice of Boolean formulas over Π ordered by implication. Let ψ be a region. The predicate abstraction of ψ with respect to the set Π of predicates is the smallest (in the implication ordering) region $\mathsf{Abs}(\psi, \Pi)$ which contains ψ and is representable as a *Boolean combination* of predicates from Π. The region $\mathsf{Abs}(\psi, \Pi)$ can be computed by recursively splitting as follows [Das et al. 1999]:

$$\mathsf{Abs}(\psi, \Pi) \doteq \begin{cases} \mathit{true} & \text{if } \Pi = \emptyset \text{ and } \psi \text{ satisfiable} \\ \mathit{false} & \text{if } \Pi = \emptyset \text{ and } \psi \text{ unsatisfiable} \\ (p \wedge \mathsf{Abs}(\psi \wedge p, \Pi')) \vee (\neg p \wedge \mathsf{Abs}(\psi \wedge \neg p, \Pi')) & \text{if } \Pi = \{p\} \cup \Pi' \end{cases}$$

The satisfiability checks can be discharged by a decision procedure [Nelson 1981; Dutertre and Moura ; de Moura and Bjørner 2008]. In the worst case, the computation is exponential in the number of predicates, and several heuristics with better performance in practice have been proposed [Saïdi and Shankar 1999; Flanagan and Qadeer 2002].

Many implementations of predicate-based software model checkers (including SLAM and Blast) implement an over-approximation of the predicate abstraction that can be computed efficiently in order to avoid the exponential cost. *Cartesian* predicate abstraction is one such precision-efficiency tradeoff: it can be computed more efficiently than full predicate abstraction but can be quite imprecise in the worst case. Cartesian abstraction formalizes the idea of ignoring relations between components of tuples, and approximates a set of tuples by the smallest Cartesian product containing the set [Ball et al. 2001]. Formally, the cartesian abstraction of ψ with respect to the set Π of predicates is the smallest (in the implication ordering) region $\mathsf{CartAbs}(\psi, \Pi)$ which contains ψ and is representable as a *conjunction* of predicates from Π. The region $\mathsf{CartAbs}(\psi, \Pi)$ can be computed as:

$$\mathsf{CartAbs}(\psi, \Pi) \doteq \begin{cases} \mathit{true} & \text{if } \Pi = \emptyset \\ p \wedge \mathsf{CartAbs}(\psi, \Pi') & \text{if } \Pi = \{p\} \cup \Pi' \text{ and } (\psi \wedge \neg p) \text{ unsatisfiable} \end{cases}$$

Cartesian predicate abstraction was implemented for C programs as part of SLAM in a tool called c2bp [Ball et al. 2001], and since then in other software verifiers.

5.4 Example: Control Abstraction

So far, we have used the abstract domain to reduce the space of data values, while keeping each path of the program precise. Thus, our algorithms so far are *path-sensitive*. In an orthogonal direction, we can develop abstract reachability algorithms in which different paths of the program are merged into equivalence classes.

For example, in a *flow-sensitive, path-insensitive* analysis, the reachability algorithm will merge the set of abstract states coming into a program location from any of its predecessor locations. In a *flow-insensitive* analysis, the ordering on program transitions is abstracted, and the program is considered as a bag of transitions which can fire in any order. Historically, model checking has always assumed path-sensitivity in the analysis, and dataflow analysis has rarely assumed path-sensitivity (or even flow-sensitivity). A uniform framework for abstract reachability analysis, which gives flow-insensitive, flow-sensitive, and path-sensitive analyses as special cases is provided in [Beyer et al. 2007].

We have primarily considered the verification of sequential programs. We can reduce the verification of non-recursive concurrent programs (*e.g.,* non-recursive multithreaded programs), to sequential programs, by taking the product of the control flow graphs of the different threads [Dwyer and Clarke 1994], and then applying the sequential analysis. However, in many cases, combinatorial explosion makes a product construction prohibitively expensive. In such cases, one approach is to perform a control abstraction for the individual threads, before analyzing the product. To do so, one can first use a data abstraction (*e.g.,* predicate abstraction, polyhedra) to compute finite-state abstractions of the individual threads, and next, apply (bi)simulation quotienting [Bouajjani et al. 1990] to obtain a small control skeleton for each thread. While the direct products may be too large to analyze, it can be feasible to analyze the products of the reduced state machines. This approach was suggested by the MAGIC software model checker [Chaki et al. 2004]. One can refine this approach by using compositional reasoning to iteratively compute an abstraction for each thread that is sound *with respect to* the behavior of the other threads, to obtain a *thread-modular* style of reasoning [Flanagan et al. 2002; Flanagan et al. 2005; Henzinger et al. 2003]. Instead of (bi)simulation quotients, one can use the L^* algorithm from machine learning to compute a small state machine that generates the observable traces for each thread [Giannakopoulou and Pasareanu 2008]. Intuitively, in each of the above cases, the small state machine computed for each thread can be viewed as a *summary* of the behavior of the thread relevant to proving the property of the concurrent program.

5.5 Combined Abstractions

Finally, one can build powerful analyses by combining several different abstractions, each designed for capturing different kind of property of the program. One way to achieve the combination is to analyze the program in two (or more) stages. In the first stage, one can use a particular abstraction (e.g. polyhedra or octagons) to compute invariants, which can be used to strengthen the abstract regions computed in the second stage. This approach, which is implemented in the F-SOFT [Jain et al. 2006] and IMPACT [McMillan 2006] model checkers, can make verification much faster if if the first phase can cheaply find invariants that are crucial for the second phase, but which are expensive to compute using a more general abstraction of the second phase. A second approach is to simultaneously combine multiple abstractions using the notion of a *reduced product* [Cousot and Cousot 1979]. The Astree analysis tool [Blanchet et al. 2002; Blanchet et al. 2003] is probably the best example of using combinations of abstract domains to enable precise and scalable verification. The software model checker

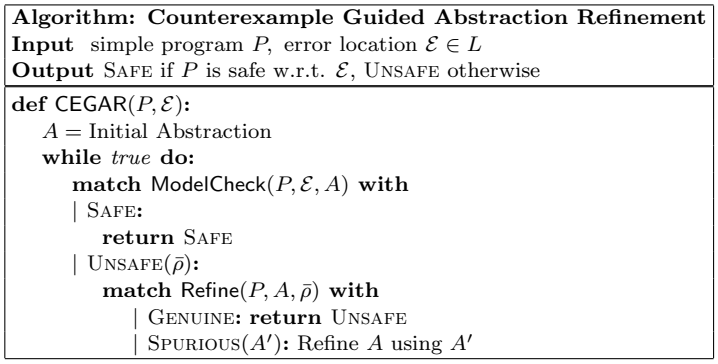

Fig. 4. Counterexample-guided Abstraction Refinement

BLAST combines predicate abstraction with arbitrary other abstractions specified via a lattice [Fischer et al. 2005; Beyer et al. 2007]. This approach allows one to extend model checkers with abstractions targeted at particular domains in a modular manner. Finally, [Gulwani and Tiwari 2006] shows a general framework for combining abstract interpretations for different theories, analogous to the manner in which decision procedures for different theories are combined.

6. ABSTRACTION REFINEMENT

In general, abstract model checking is *sound*, *i.e.,* programs proved to be safe by the abstract analysis are actually safe, but *incomplete*, *i.e.,* the abstract analysis can return a counterexample even though the program is safe. In case the abstract analysis produces a counterexample, we would like to design techniques that determine whether the counterexample is *genuine*, *i.e.,* can be reproduced on the concrete program, or *spurious*, *i.e.,* does not correspond to a real computation but arises due to imprecisions in the analysis. In the latter case, we would additionally like to *automatically refine* the abstract domain, that is, construct a new abstract domain that can represent strictly more sets of concrete program states. The intent of the refined domain is to provide a more precise analysis which rules out the current counterexample and possibly others. This iterative strategy was proposed as *localization reduction* in [Kurshan 1994; Alur et al. 1995] and generalized to *counterexample-guided refinement* (CEGAR) in [Ball and Rajamani 2000b; Clarke et al. 2000; Saidi 2000]. Figure 4 formalizes this iterative refinement strategy in procedure CEGAR, which takes as input a program P, and error location $\mathcal{E}$ and an initial, possibly trivial, abstract domain A. The procedure iteratively constructs refinements of the abstract domain A until either it suffices to prove the program safe, or the procedure finds a genuine counterexample.

6.1 Counterexamples and Refinement

The most common form of counterexample-guided refinement in software model checking has the following ingredients. The input to the counterexample analysis algorithm is a path in the control flow graph ending in the error location. The path represents a possible counterexample produced by abstract reachability analysis.

```
0:  LOCK = 0;
1:  do {
      LOCK = 1;
      old = new;
2:    if (*) {
3:      LOCK = 0;
        new++;
      }
4:  } while (new != old);
5:  if (LOCK==0)
6:    error();
    LOCK = 0;
```

Fig. 5. Program

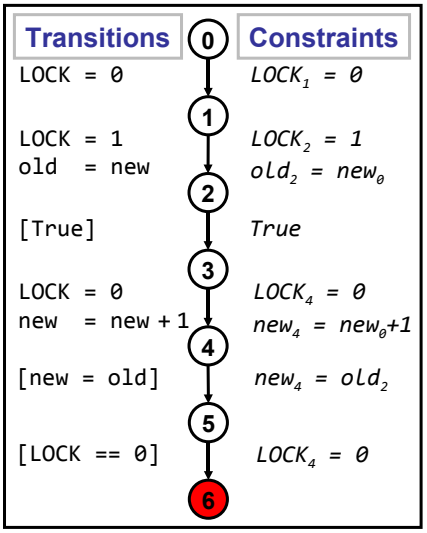

Fig. 6. Abstract Counterexample

The first step of the algorithm constructs a logical formula, called the *trace formula*, from the path, such that the formula is satisfiable iff the path is executable by the concrete program. Second, a decision procedure is used to check if the trace formula is satisfiable. If satisfiable, the path is reported as a concrete counterexample to the property. If not, the proof of unsatisfiability is mined for new predicates that can rule out the current counterexample when the abstract domain is augmented with these predicates. The CEGAR loop makes progress by eliminating at least one counterexample in each step. Since each iteration refines the abstract domain from the previous iteration, this guarantees that all previously excluded counterexamples remain excluded in the next iteration.

Counterexamples. An *abstract counterexample* $\bar{\rho}$ for a program P is a path

$$\ell_0 \xrightarrow{\rho_0} \ell_1 \ldots \xrightarrow{\rho_{n-1}} \ell_n \tag{1}$$

of P where ℓ_0 is the initial location and ℓ_n is the error location.

Consider again the program from Figure 5, and an abstract reachability analysis using only the predicates `LOCK` $= 0$ and `LOCK` $= 1$. The abstract reachability analysis can return a counterexample path like the one shown in Figure 6. The vertices correspond to the labels ℓ and the edges to transitions ρ. To the left of each edge we write the program operation corresponding to the transition. This counterexample is spurious, that is, does not correspond to a concrete program execution. Intuitively, this is because after the second transition the variables `1` and `old` are equal, after the fourth transition, where `new` is incremented they are disequal, and so, it is not possible to break out of the loop as happens in the fifth transition. The abstract model checking does not track the equality of `new` and `old` and hence admits this spurious path.

Trace Formulas. To convert an abstract counterexample into a trace formula, we

rename the state variables at each transition of the counterexample and conjoin the resulting transition constraints to get the following formula:

$$\bigwedge_{i=0}^{n-1} \rho_i(X_i, X_{i+1}) \tag{2}$$

Notice that the trace formula is equivalent to:

—the bounded model checking formula for the unrolled version of the program corresponding to the path,

—the strongest postcondition of *true* with respect to the straight-line program corresponding to the path, and

—the weakest precondition of *true* with respect to the straight-line program corresponding to the path.

Thus, the trace formula is satisfiable iff the path is executable. To avoid constraints of the form $x_{i+1} = x_i$ for each x not modified by an operation, we can convert the path to *static single-assignment (SSA)* form [Flanagan and Saxe 2000; Henzinger et al. 2004], after which the renamed operations directly get translated into constraints. The trace formula can be further optimized by statically slicing out parts of the path that are not relevant to the reachability of the error states [Jhala and Majumdar 2005].

Figure 6 shows the individual constraints of the trace formula on the right side of the trace. The names $\texttt{LOCK}_i$ refer to the different values of the (state) variable `LOCK` at different points along the trace. We have used the SSA optimization – the constraint corresponding to the incrementing of `new` stipulates that the incremented value $\texttt{new}_4$ is one greater than the previous value $\texttt{new}_0$.

Syntax-based Refinement. Suppose the trace formula is unsatisfiable. One simple way to deduce new predicates that are sufficient to rule out the current spurious counterexample is to find an *unsatisfiable core* of atomic predicates appearing in the formula, whose conjuction is inconsistent. There are several ways of finding such a set. First, one can use a greedy algorithm to find a *minimal* set of constraints that is inconsistent [Ball and Rajamani 2002a]. Second, one can query a proof producing decision procedure [Necula and Lee 2000] to find a proof of unsatisfiability of the constraints, and choose the atomic formulas that appear as the leaves in this proof. After finding the atomic predicates, we can simply drop the subscripts and add the resulting predicates to the tracked set, thereby refining the abstraction.

Consider the trace shown in Figure 6. The trace formula, given by the conjunction of the constraints on the right side of the trace, is unsatisfiable, as it contains the conjunction of

$$\texttt{old}_2 = \texttt{new}_0, \quad \texttt{new}_4 = \texttt{new}_0 + 1, \quad \texttt{new}_4 = \texttt{old}_2$$

which is inconsistent. Thus, by dropping the subscripts, we can refine the abstraction with the new predicates

$$\texttt{old} = \texttt{new}, \quad \texttt{new} = \texttt{new} + 1, \quad \texttt{new} = \texttt{old}$$

which, after dropping redundant and inconsistent predicates, leaves just the predicate $\texttt{new} = \texttt{old}$. Notice that when this predicate is added to the set of predicates,

the resulting set, namely

$$\{\texttt{LOCK} = 0, \quad \texttt{LOCK} = 1, \quad \texttt{new} = \texttt{old}\}$$

suffices to refute the counterexample. That is, the given path is *not* a counterexample in the abstract model generated from these predicates.

Finally, another syntax-based refinement strategy is to bypass the trace formula construction, and instead, compute the the sequence of predecessors

$$\varphi_n = \mathit{true} \qquad \varphi_{i-1} = \mathit{Pre}(\varphi_i, \rho_i)$$

along the counterexample path, starting at the error location and going all the way back to the initial location. The abstraction can then be refined by adding the atomic predicates appearing in each φ_i. This technique was proposed by [Namjoshi and Kurshan 2000] and is used in [Ivancic et al. 2008].

Interpolation-based Refinement. Though the syntax-based methods suffice to eliminate a particular counterexample, they are limited by the fact that they essentially capture relationships that are *explicit* in the program text, but can miss relationships that are *implicit*. An alternate refinement strategy, suggested in [Henzinger et al. 2004], uses *Craig Interpolation* to find predicates that capture the implicit relationships that are required to verify a given safety property.

Let ϕ^- and ϕ^+ be two formulas whose conjunction is unsatisfiable. An *interpolant* ψ for (ϕ^-, ϕ^+) is a formula such that (a) ϕ^- implies ψ, (b) $\psi \wedge \phi^+$ is unsatisfiable, and (c) the free variables in ψ are a subset of the free variables that are common to ϕ^- and ϕ^+. An interpolant always exists in case (ϕ^-, ϕ^+) are first-order formulas [Craig 1957], and an interpolant can be constructed in time linear in the size of a resolution proof for formulas in the combination theories of equality with uninterpreted functions and linear arithmetic [McMillan 2004].

Now consider an unsatisfiable trace formula (2), and for each $j \in \{0, \ldots, n-1\}$, consider the *j-cut*:

$$(\bigwedge_{i=0}^{j} \rho_i(X_i, X_{i+1}), \bigwedge_{i=j+1}^{n-1} \rho_i(X_i, X_{i+1}))$$

Clearly, the conjunction of the two formulas (the trace formula) is unsatisfiable. Also, the common variables between the two formulas of the cut are from X_{j+1}. Intuitively, the first part of the cut defines the set of states that can be reached by executing the prefix of the counterexample trace up to step j, and the second part defines the set of states that can execute the suffix. The common variables between the two parts are variables that are *live* across step j, *i.e.,* defined in the prefix and used in the suffix. Thus, an interpolant for the j-cut (a) contains states that are reached by executing the prefix, (b) cannot execute the complete suffix, and (c) contain only live variables. Thus, the interpolant serves as an abstraction at step j which is enough to rule out the counterexample. Now suppose we compute interpolants I_j for each j-cut which additionally satisfy an *inductiveness condition* $I_j(X) \wedge \rho_j(X, X') \rightarrow I_{j+1}(X')$. Then, adding predicates appearing in the interpolants and performing abstract reachability with these predicates is enough to rule out the counterexample. The counterexample refinement technique in [Henzinger et al. 2004] computes the cut for each j and compute interpolants

for each j-cut. By using the same proof in the construction of interpolants, the procedure additionally ensures the inductiveness condition on interpolants.

The advantage of interpolation as a technique for refinement is that it not only discovers new predicates, but also determines the control locations at which these predicates are useful. Thus, instead of keeping a global set of predicates, one can keep a map from locations to sets of predicates, and perform predicate abstraction with respect to the local set of predicates. In experiments on device drivers (reported in [Henzinger et al. 2004]) this locality results in an order of magnitude improvement in the running times of the CEGAR loop.

Consider the trace formula from Figure 6. The 3-cut of the trace formula corresponds to the pair of formulas

$$\begin{aligned}\phi^- \doteq\ & \texttt{LOCK}_1 = 0 \wedge \\ & \texttt{LOCK}_2 = 1 \ \wedge \texttt{old}_2 = \texttt{new}_0 \wedge \\ & \mathit{true} \\ & \texttt{LOCK}_4 = 0 \wedge \texttt{new}_4 = \texttt{new}_0 + 1\end{aligned}$$

$$\begin{aligned}\phi^+ \doteq\ & \texttt{new}_4 = \texttt{old}_2 \wedge \\ & \texttt{LOCK}_4 = 0\end{aligned}$$

The variables that are common to ϕ^- and ϕ^+ are $\texttt{new}_4$, $\texttt{old}_2$ and $\texttt{LOCK}_4$, which are the SSA renamed versions of `new`, `old` and `LOCK` that are live at the cut label `4 :`. One interpolant for this cut is

$$\psi \doteq \texttt{new}_4 \neq \texttt{old}_2$$

Note that this interpolant is over the common variables, is implied by ϕ^- and is inconsistent with ϕ^+. Indeed this interpolant captures exactly the key relationship that holds at label `4` that is needed to prove safety. The interpolant yields the predicate $\texttt{new} \neq \texttt{old}$ at label `4`. Similarly, the i-cuts for $0 \leq i \leq 5$, we get the interpolants ψ_i where

$$\begin{aligned}&\psi_0 \doteq \mathit{true} && \psi_1 \doteq \mathit{true} \\ &\psi_2 \doteq \texttt{old}_2 = \texttt{new}_0 && \psi_3 \doteq \texttt{old}_2 = \texttt{new}_0 \\ &\psi_4 \doteq \texttt{old}_2 \neq \texttt{new}_0 && \psi_5 \doteq \mathit{false}\end{aligned}$$

and hence, by using the predicate $\texttt{old} = \texttt{new}$ at locations `2 :`, `3 :` and `4 :` only, we can prove the program safe.

Relative Completeness. The term *relative completeness* refers to the property that iterative counterexample refinement converges, given that there exist program invariants in a restricted language which are sufficient to prove the property of interest.

Ensuring relative completeness is not trivial. Consider the example shown in Figure 7. To verify that the error is not reachable, we must infer the invariant that $\texttt{x} = \texttt{y}$ and $\texttt{x} \geq 0$. Unlike the program in Figure 5, neither of these facts appears syntactically in the program. Figure 8 shows the (spurious) counterexample for the abstract model generated from the predicates $\texttt{x} = 0$ and $\texttt{y} = 0$. This counterexample corresponds to the unrolling of the loops once, and syntax based

```
1:  x = 0;
    y = 0;
2:  while (*){
3:    x++;
      y++;
    }
4:  while (x>0){
5:    x--;
      y--;
    }
6:  if (y != 0) error()
```

Fig. 7. Program

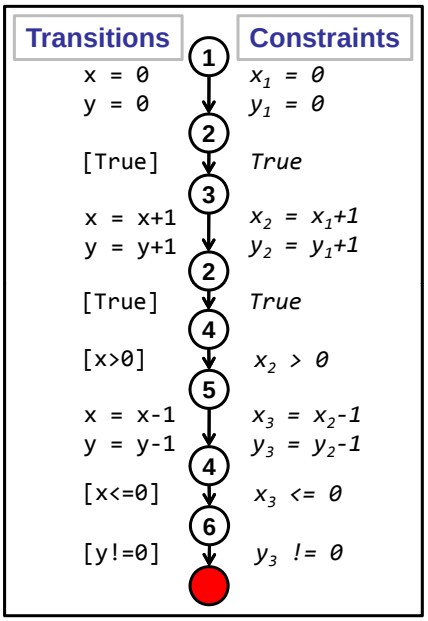

Fig. 8. Abstract Counterexample

refinement strategies return the new predicates $\mathtt{x} = 1$ and $\mathtt{y} = 1$. In general, the refinement step can diverge, generating the sequence of predicates

$$\begin{array}{ll} \mathtt{x} = 0, & \mathtt{y} = 0 \\ \mathtt{x} = 1, & \mathtt{y} = 1 \\ \mathtt{x} = 2, & \mathtt{y} = 2 \\ \vdots & \vdots \end{array}$$

for counterexamples where the loops are unrolled $0, 1, 2, \ldots$ times.

For syntactic techniques [Ball et al. 2002] provides a relative completeness result using a non-deterministic strategy to generalize from particular counterexamples. In practice, the strategy is implemented (*e.g.,* in the SLAM model checker) using heuristics specific to the problem domain.

Relative completeness can be ensured for interpolation-based techniques by *restricting* the language from which predicates are drawn, [Jhala and McMillan 2006]. For example, one can stratify the language of predicates $\mathcal{L}$ into $\mathcal{L}_0 \subset \mathcal{L}_1 \ldots$ such that $\mathcal{L} = \cup_i \mathcal{L}_i$. Next, we restrict the interpolation to find predicates in $\mathcal{L}_i$ iff there are no predicates in $\mathcal{L}_{i-1}$ that refute the counterexample. In this way, the iterative refinement becomes relatively complete in the sense that if there is some safety proof where the invariants are drawn from $\mathcal{L}$, then there is some $\mathcal{L}_j$ from which the invariants are drawn, and the restricted iterative refinement will be guaranteed to terminate by finding predicates from $\mathcal{L}_j$. One way to stratify the language is to let $\mathcal{L}_i$ be the language of predicates where the magnitude of each constant is less than i. In essence, this stratification biases the iterative refinement loop to find predicates

involving "small constants". [Jhala and McMillan 2006] shows how to structure decision procedures to force them to produce interpolants from a restricted language. [Rybalchenko and Sofronie-Stokkermans 2007] shows a more general technique for computing interpolants that satisfy different kinds of constraints.

In the example of Figure 7, the restricted interpolation method directly finds the predicates $\mathtt{x} = \mathtt{y}$ and $\mathtt{x} \geq 0$ which belong in $\mathcal{L}_0$, the language of predicates with constants less than 0. This simple restriction goes a long way towards making CEGAR complete.

Refining Multiple Paths. Several augmentations to the above refinement scheme have been suggested. First, instead of refining one path at a time, the procedure can be called with a set of abstract counterexamples which are refined together while optimizing the set of inferred predicates [Chaki et al. 2003]. In [Beyer et al. 2007], counterexample analysis is performed on a *path program*, the least syntactically valid sub-program containing the counterexample. A path program represents a (possibly infinite) family of possible counterexamples. The goal of refining path programs (through *path invariants*) is to find suitable program invariants that simultaneously rule out the entire family. Unfortunately, since a path program can contain loops, the simple partitioning and interpolation technique from above is not immediately applicable. In [Beyer et al. 2007], counterexample refinement is performed by inferring path invariants using constraint-based invariant synthesis.

Refining Other Domains. While we have focussed on refining predicate abstractions by adding new predicates, the idea of counterexample refinement can be used for other abstract domains. For example, [Jhala and McMillan 2005] shows how *symmetric interpolation* can be used to refine (propositional) transition relations in a counterexample guided manner, [Gulavani et al. 2008] shows how a polyhedral domain can be refined using counterexample analysis. For control abstractions in thread-modular reasoning, [Henzinger et al. 2004] show how environment assumptions for concurrent programs can be automatically constructed and refined.

Refinement techniques have been generalized to domains beyond hardware and software: e.g., [Alur et al. 2006; Jha et al. 2007] apply the technique in verifying real-time and hybrid systems.

6.2 Abstraction-Refinement based Model Checkers

Slam. The SLAM model checker [Ball and Rajamani 2002b] was the first implementation of CEGAR for C programs. It introduced *Boolean programs* — imperative programs where each variable is Boolean — as an intermediate language to represent program abstractions. A tool (called `c2bp`) implemented predicate abstraction for C programs [Ball et al. 2001]. The input to `c2bp` is a C program and a set of predicates, and the output is a Boolean program, where each Boolean variable corresponds to a predicate, and the assignments and conditionals on the variables correspond to the Cartesian predicate abstraction of the C program. A tool called BEBOP then implemented a symbolic model checker for (recursive) Boolean programs [Ball and Rajamani 2000a]. Finally, abstraction refinement was performed by `newton`, which implemented a greedy heuristic to infer new predicates from the trace formula. SLAM was used successfully within Microsoft for device driver verification [Ball et al. 2006] and developed into a commercial product (Static Driver

Verifier, SDV).

The SLAM project introduced several key ideas in software model checking, including the generalization of predicate abstraction in the presence of pointers and dynamically allocated memory [Ball et al. 2001], modular predicate abstraction [Ball et al. 2005], and BDD-based model checking in the presence of procedure calls [Ball and Rajamani 2000a]. SLAM inspired a resurgence of interest in the verification of software implementations and a suite of tools geared to program verification.

Blast. The BLAST model checker [Beyer et al. 2007] implements an optimization of CEGAR called *lazy abstraction*. The core idea of BLAST is the observation that the computationally intensive steps of abstraction and refinement can be optimized by a tighter integration which would enable the reuse of work performed in one iteration in subsequent iterations. Lazy abstraction [Henzinger et al. 2002] tightly couples abstraction and refinement by constructing the abstract model *on-the-fly*, and locally refining the model *on-demand*. The former eliminates an often wasteful and expensive model-construction phase and instead, performs abstraction on the *reachable* part of the state space. This is achieved by lazily building an *abstract reachability tree* whose nodes are labeled by abstract regions. The regions at different nodes of the tree, and hence, at different parts of the state space, can be over different sets of predicates. To locally refine the search when a counterexample is found, BLAST finds the "pivot" node from which the remainder of the counterexample is infeasible and rebuilds the subtree from the pivot node onwards. This ensures that parts of the state-space known to be free of errors, namely different subtrees, are not re-analyzed. Further, this permits the use of different predicates at different program points which drastically reduces the size of the abstract state space that must be analyzed. Upon termination with the outcome "program correct," the proof is not an abstract model on a global set of predicates, but an abstract model whose predicates change from state to state. Thus, by always maintaining the minimal necessary information to validate or invalidate the property, lazy abstraction scales to large systems without sacrificing precision. BLAST combines lazy abstraction with procedure summaries [Reps et al. 1995] and scoped interpolation [Henzinger et al. 2004] to model check recursive programs. Finally, BLAST can be extended with arbitrary lattices [Fischer et al. 2005], thus yielding a general mechanism for refining any dataflow analysis with iterative predicate abstraction.

Magic. The MAGIC model checker was designed to enable the modular verification of concurrent, message passing C programs. MAGIC allows the user to specify arbitrary non-deterministic labeled transition systems (LTS) as specifications, and verifies that the set of traces (over messages and events) generated by a concurrent C program is contained in in the language generated by the LTS. MAGIC implements a *two-level* abstraction strategy to combat the state explosion that arises from the product of multiple threads. First, an (eager) predicate abstraction is carried out for each individual thread, yielding a finite state machine representing the thread's behavior. Second, an *action-guided* abstraction is carried out to minimize each thread's state machine while preserving the sequences of messages and events generated by the state machine. Finally, the product of the reduced state machines is computed and model checked. The entire procedure is wrapped inside a CEGAR loop, that uses spurious counterexamples to infer new predicates which yield refined

(reduced) state machines. Finally, MAGIC also implements several methods to minimize the number of predicates, by finding predicates that simultaneously refute multiple paths [Chaki et al. 2003].

F-Soft. The F-SOFT model checker [Ivancic et al. 2005] combines CEGAR-based predicate abstraction refinement with several other abstract domains that efficiently yield the kinds of invariants needed to check standard runtime errors in C programs (e.g. buffer overflows, null dereferences). For these errors, a CEGAR based predicate abstraction can eventually find the right domain, but a multi-stage framework that eagerly combines numerical domains with CEGAR can be much more efficient. The numerical domains can altogether eliminate some easy checks, or yield invariants that help the subsequent iterative analysis converge more quickly [Jain et al. 2006]. To this end F-SOFT implements symbolic model checking algorithms that combine BDDs and polyhedra [Yang et al. 2006], and techniques that combine widening with iterative refinement [Wang et al. 2007].

Impact. The IMPACT model checker [McMillan 2006] implements an even lazier variant of lazy abstraction, that entirely eliminates all abstract (post) image computations. Instead IMPACT builds an abstract reachability tree by directly using the sequence of interpolants generated from the trace formulas to strengthen the regions labeling the nodes of the tree. This process is repeated using the lazy abstraction paradigm until the program is proved safe, or a counterexample is found. [McMillan 2006] shows that by directly using interpolation within the lazy abstraction framework, one can achieve very dramatic improvements in performance.

7. PROCEDURAL ABSTRACTION

♣ TO BE UPDATED♣

So far, we have considered programs without procedures. We now extend the class of simple programs imp to the class imp+proc of programs with (potentially recursive) procedure calls.

7.1 Programs with Procedures

A *procedural imperative program* P is a tuple $(\mathtt{F}, \mathtt{f0})$ where $\mathtt{F}$ is a finite set of procedures, and $\mathtt{f0}$ is a special *initial* procedure in $\mathtt{F}$ from which execution begins. A *procedure* $\mathtt{f}$ is a simple imperative program $(\ell^{\mathtt{f}}, X^{\mathtt{f}}, \ell_0^{\mathtt{f}}, T^{\mathtt{f}})$. Each procedure $\mathtt{f}$ has a unique *input parameter* $x_0^{\mathtt{f}}$ in $X^{\mathtt{f}}$. The states of a procedure $\mathtt{f}$ are the elements of $v.X^{\mathtt{f}}$. The set of transitions comprises:

—**Intraprocedural** transitions of the form $(\ell, \mathsf{Intra}\ \rho, \ell')$,

—**Call** transitions of the form $(\ell, \mathsf{Call}\ \mathtt{x} \leftarrow \mathtt{f}(e), \ell')$, and,

—**Return** transitions of the form $(\ell, \mathsf{Ret}\ \mathtt{return}\ e, \ell')$.

For simplicity, we assume that the set of variables of each procedure are distinct – that is each procedure has its own *local* variables and there are no global variables. Further, we assume for simplicity, that the formal parameters of each procedure are *not modified* by the transitions of the procedure. We write ℓ to abbreviate $\cup_{\mathtt{f}}\ell^{\mathtt{f}}$, X to abbreviate $\cup_{\mathtt{f}}\ell^{\mathtt{f}}$, T to abbreviate $\cup_{\mathtt{f}}T^{\mathtt{f}}$.

7.2 Interprocedural Reachability

Even if all the variables X range over finite domains, *e.g.,* are Boolean valued, the state space of a procedural program is infinite, as the stack can grow unboundedly due to recursive calls. As a result, we cannot use direct graph reachability (as in Algorithm 1) to verify safety properties of programs with procedures. The key to analyzing procedural programs lies in designing an algorithm that uses the following two observations to obviate the need for tracking the control stack. First, the behaviors of the entire program can be reconstituted from the *input-output* behaviors of the individual procedures. Second, even though the number of configurations of the whole program are infinite, each procedure has a *finite* number of input-output behaviors.

To use these observations we extend the standard reachability procedure with a form of *memoization* thus equipping it to compute input-output *summaries* for each procedure. If the all the program variables have a finite domain, then the size of the summaries – which are simply sets of pairs of input parameter values and output expression values – is finite. Consequently, the extended procedure can compute the summaries as a least fixpoint, thereby yielding an algorithm for model checking programs with procedures. Thus to compute summaries, we extend the reachability procedure from from Figure 1 with the following data structures.

(1) An *input* state for a procedure `f` is a valuation where all the variables the formal parameter $\mathtt{x}_0^{\mathtt{f}}$ are set to 0. The *input of a state* $s \in v.X^{\mathtt{f}}$, written $\mathsf{Init}(s)$, is the input state for `f` where the formal $\mathtt{x}_0^{\mathtt{f}}$ has the value $s(\mathtt{x}_0^{\mathtt{f}})$. As the formal parameters $\mathtt{x}_0^{\mathtt{f}}$ of each procedure remain unmodified, each state has encoded within it the input with which the corresponding procedure was called.

(2) The *callers* of an input state s of `f`, written $\mathit{Callers}[s]$, correspond to tuples $(s_c, \mathsf{Call}\ op_c, \ell'_c)$ such that there is a reachable configuration (ℓ_c, s_c) and a call transition (ℓ_c, op_c, ℓ'_c) which when executed from state s_c yields the input state s for `f`.

(3) The *exits* of an input state s of `f`, written $\mathit{Exits}[s]$, correspond to pairs (s_e, op_e) such that from the input configuration $(\ell_0^{\mathtt{f}}, s)$ some configuration (ℓ, s_e) is reachable, and there is a return transition $(\ell, \mathtt{return\ e}, \cdot)$ in `f`.

Intuitively, the callers and exits allow us to build a memoized version of the enumerative reachability algorithm. Whenever a call transition is reached, the known exits for the corresponding input state are propagated at the caller. Whenever a return transition is reached, the return value is propagated to all the known callers of the corresponding input state.

This intuition is formalized in Figure 9 which shows the interprocedural reachability algorithm that extends Algorithm Reachability to handle procedure calls and returns. InterProceduralReachability is a worklist-based algorithm that simultaneously computes the reachable input states for each procedure, the callers of each input state and the exits of each input state. For each procedure, the least fixpoint set of input states corresponds to all the (finitely many) reachable inputs of the procedure, and the respective callers and exits correspond to the possible calling states and return values for that input state.

The algorithm maintains a worklist of configurations, namely pairs of locations

and states. At each iteration, a is picked off the worklist. If the configuration has previously been reached, the algorithm moves to the next worklist element. If the configuration is new, its successors are added to the worklist as follows.

—For each enabled intraprocedural transition for the configuration, the successors are computed using *Post*, and added to the worklist, similar to Algorithm Reachability.

—For each enabled call transition of the form $\mathtt{x} \leftarrow \mathtt{f}(\mathtt{e})$ the algorithm carries out the following steps. First, it computes the input state for the call, by calling post with the transition that assigns the formal $\mathtt{x}_0^{\mathtt{f}}$ the actual parameter `e`. Second, it adds the calling state to the set of known callers for the input state. Third, each known *exit* for the input state, is *propagated* to the caller by adding to the worklist the succesors corresponding to assigning the return values at the exit to `x`.

—For each enabled return transition of the form `return e` the algorithm carries out the following steps. First, it computes the input state for the exit state from which the return occurs. Second, it adds the exit state to the set of known exits for the input state. Third, it propagates the return value at the exit state to each of the known callers of the input state.

We can prove that this algorithm computes all the reachable configurations by induction on the length of the shortest trace leading to the configuration. We can prove that the algorithm terminates by appealing to the fact that the sets being populated, namely the reach set, callers and exits are all finite.

Graph-based Algorithms. Algorithm InterProceduralReachability is a reformulation of the classical tabulation-based approach to interprocedural dataflow analysis presented in [Sharir and Pnueli 1981]. [Reps et al. 1995] showed how the technique could be phrased as a special graph-reachability algorithm, leading to an algorithm with running time cubic in the number of reachable configurations. Recently [Chaudhuri 2008] has shown how using bitvectors to represent sets of configurations and bitvector operations to manipulate the sets, one can obtain a sub-cubic running time.

The idea of performing reachability on a graph while ensuring the path defines a word in a context free language has also been used in the context of Datalog query evaluation [Yannakakis 1990], and the approach ultimately uses a dynamic programming algorithm for parsing context free languages.

Symbolic Algorithms. As in the case of (simple) explicit state model checking, the above enumerative algorithms run in time that is polynomial in the number of reachable configurations. However, assuming that each variable can take on at least two values, the number of reachable configurations is *exponential* in the number of local variables of each procedure, and hence, exponential in the size of the program's representation.

However, the technique of summarization can be used to lift symbolic algorithms, which work with compact representations of *sets* of states, to the interprocedural setting. For example, to obtain a symbolic version of InterProceduralReachability, we need only to:

(1) view each s as a set of states,

Algorithm: Interprocedural Enumerative Reachability
Input simple program $P = (X, L, T, \ell_0, F, \mathtt{f0})$, error location $\mathcal{E} \in L$
Output SAFE if P is safe w.r.t. $\mathcal{E}$, UNSAFE otherwise

```
def Propagate(s_c, s_e, x ← f(·), return e, ℓ'_c):
    add (ℓ'_c, Post(s_c, ⟨x ← e(s_e)⟩)) to worklist

def PropagateCall(s_c, · ← f(e) as op_c, ℓ'_c):
    s' = Post(s_0^f, ⟨x_0^f ← e(s_c)⟩)
    add (ℓ_0^f, s') to worklist
    add (s_c, op_c, ℓ'_c) to Callers[Init(s')]
    foreach (s_e, op_e) in Exits[Init(s')] do:
        Propagate(s_c, s_e, op_c, op_e, ℓ'_c)

def PropagateReturn(s_e, op_e):
    add (s_e, op_e) to Exits[Init(s_e)]
    foreach (s_c, op_c, ℓ'_c) in Callers[Init(s_e)] do:
        Propagate(s_c, s_e, op_c, op_e, ℓ'_c)

def InterProceduralReachability(P, E):
    reach = ∅
    worklist = {(ℓ_0^f0, s) | s ∈ v.X^f0}
    while worklist ≠ ∅ do:
        pick and remove(ℓ, s) from worklist
        if (ℓ, s) ∉ reach:
            add (ℓ, s) to reach
            foreach (ℓ, Intra ρ, ℓ') in T do:
                add {(ℓ', s') | s' ∈ Post(s, ρ)} to worklist
            foreach (ℓ, Call op, ℓ') in T do:
                PropagateCall(s, op, ℓ')
            foreach (ℓ, Ret op, ℓ') in T do:
                PropagateReturn(s, op, ℓ')
    if exists (E, s) ∈ reach:
        return UNSAFE
    else:
    return SAFE
```

Fig. 9. Enumerative Interprocedural Reachability

(2) view the membership in the reach set as inclusion in the reach set,

(3) view each *Post* operation as a symbolic operation, as in Section 4.

[Ball and Rajamani 2000a] describes BEBOP, the first BDD-based interprocedural safety model checker that takes as input a *Boolean program*, that is a procedural program where each variable is Boolean valued, and a distinguished error location and determines whether the error location is reachable in the Boolean program. In addition to describing how to marry procedure summaries with BDDs, they describe how to generate *counterexample* traces from the runs of the checker.

Abstraction. So far we have considered programs where the variables take on finitely many values.

We can use abstraction to extend the finite-state approach to programs with variables over infinite domains. As for programs in imp, there are two approaches. The first is to *eagerly* abstract the program into a finite state procedural program that overapproximates the behaviors of the original pro-

gram. This method was popularized by SLAM which uses predicate abstraction [Agerwala and Misra 1978; Graf and Saïdi 1997] to abstract C programs into Boolean programs [Ball et al. 2001] which are subsequently model checked using BEBOPas described above. The second is to *lazily* carry out the abstraction *during* the model checking. For example, to obtain an abstract version of InterProceduralReachability, we need only to:

(1) view each s as an abstract set of states,
(2) view the membership in the reach set as inclusion in the abstract reach set,
(3) view each *Post* operation as an overapproximate $Post^{\#}$ operation, as in Section 5.

This is the approach adopted by BLAST, which uses predicate abstraction to lazily build abstract summaries.

In either case, to ensure soundness, the abstract states constructed when analyzing a procedure `f` must describe valuations of *only* those variables which are in scope inside procedure `f`. For example, when predicate abstraction is used eagerly or lazily, to model check procedural programs, we must ensure that all the predicates used to abstract procedure `f` are *well-scoped*, *i.e.,* are local variables of `f`.

The efficiency of the model checking is greatly enhanced by using abstractions that yield summaries that can be applied at multiple callsites, instead of individual summaries that can only be applied at specific call-sites. For example, it is typically more efficient to use *relational abstractions* [Cousot and Cousot 1979] that can describe the outputs in terms of the inputs, instead of pairs of input-output states, where each pair corresponds to a different callsite. For example, a relational summary that specifies that the output of a procedure is one greater than the input passed to the procedure, is more compact and reusable, than a tabulational summary that specifies that if the input is 0 (resp. 1, 2), the output is 1 (resp. 2, 3).

[Ball et al. 2005] shows how relational summaries can be computed by using predicate abstraction over predicates containing *symbolic constants*, which are immutable values representing the input parameters of different functions. Algorithm InterProceduralReachability exploits this trick as well, by requiring that the formals remain unmodified. As a result each concrete output state (*e.g.,* $[\mathtt{x}_0^{\mathtt{f}} \mapsto 0, \mathtt{ret} \mapsto 1]$) implicitly encodes the input state that generated it, and hence a relational abstraction of the output state (*e.g.,* $\mathtt{ret} = \mathtt{x}_0^{\mathtt{f}} + 1$) describes an input-output behavior of the function. [Henzinger et al. 2004] shows how Craig Interpolation can be used to address the problem of *automatically discovering* well-scoped predicates with symbolic constants, that are relevant to verifying a particular safety property.

Top-Down vs. Bottom-Up. The algorithm described above computes the set of reachable inputs for each procedure in a *top-down* manner, that is, using the callsites. Another approach is to aggressively compute the most general behavior for each procedure, by seeding the worklist with all the possible input states for each procedure. This *bottom-up* approach suffers from the disadvantage of computing possibly useless information. On the other hand, it enjoys several engineering

advantages. The procedure call graph, is a graph whose vertices are procedures, and which has directed edges from $\texttt{f}_1$ to $\texttt{f}_2$ if there is a call transition in $\texttt{f}_1$ to procedure $\texttt{f}_2$. Suppose that there is no recursion, and so the procedure call graph is acyclic. First, we can analyze each procedure in isolation, processing the leaf procedures of the call-graph first and moving up the graph. This can drastically lower the memory needed to analyze massive programs. Second, as procedures can be analyzed in isolation, we can parallelize the analysis as we can concurrently analyze two procedures each of whose callees has been previously analyzed. Bottom-up analyses typically use relational abstractions, as when analyzing a procedure `f`, no information is available about the states from which `f` can be called.

SATURN is a bottom-up interprocedural model checker that exploits the engineering advantages outlined above to scale to the entire Linux kernel [Xie and Aiken 2005]. SATURN computes input-output summaries over a fixed set of predicates as follows. Each input or output configuration is a conjunction of the predicates. SATURN unfolds the transition relation for the body of the procedure (plugging in the summaries for called procedures), and then uses SAT-based Bounded Model Checking to determine which input-output configurations are feasible. The summary computed for the procedure is the *disjunction* of all the satisfiable configurations.

HOUDINI embodies a dual approach for inferring pre- and post-conditions for procedures [Flanagan et al. 2001]. Intuitively, the input-output behavior of each procedure can be summarized by the pair of the conjunction of all precondition clauses, and, the conjunction of all postcondition clauses. HOUDINI generates pre- and post-conditions from a candidate set of predicates as follows. It begins with summaries corresponding to the conjunction of *all* predicates, and then uses SMT-based Bounded Model Checking (ESC/JAVA [Flanagan et al. 2002]) to iteratively drop the precondition (resp. postcondition) predicates that do not provably hold at some call (resp. exit) location. It can be shown that the above converges to a least fixpoint, where each summary's precondition (resp. postcondition) captures the *strongest* overapproximation of the input states (resp. exit states) for the procedure, expressible as a conjunction of the candidate predicates.

7.3 Concurrency and Recursion

For concurrent programs where each sequential thread belongs to the class imp+proc, the reachability problem is undecidable [Ramalingam 2000]. The intuition is that the execution of a procedural program can be modeled as a context free language over the alphabet of local and synchronization actions (and indeed each context free language can be obtained in this way), and a configuration (c_1, c_2) of a concurrent program with two threads can be reached iff the intersection $L_1 \cap L_2$ of two context free languages over the alphabet of synchronization actions is non-empty. Conversely, as shown in [Ramalingam 2000], for any two context free languages L_1 and L_2, one can build a program $P_1 | \, | P_2$ synchronizing over the common alphabet of L_1 and L_2 and a configuration (c_1, c_2) of $P_1 | \, | P_2$ such that $L_1 \cap L_2$ is non-empty iff (c_1, c_2) is reachable. This shows the problem is undecidable.

Several techniques to over-approximate the set of reachable states have been suggested [Bouajjani et al. 2003; Chaki et al. 2006]. These techniques relax the synchronization sequences possible in the original program, *e.g.,* by ignoring the

ordering of synchronization operations [Bouajjani et al. 2003], or by approximating the context free language by a regular one [Chaki et al. 2006]. Using the notion of *transactions* (sequences of actions of a thread that execute atomically w.r.t. other threads), [Qadeer et al. 2004] show a summarization-based model checking algorithm for concurrent procedural programs which is sound but incomplete. In special cases, such as concurrent threads synchronizing solely using nested locks, the reachability problem is, somewhat surprisingly, decidable [Kahlon and Gupta 2007].

The reachability problem for concurrent procedural programs becomes decidable if one bounds the number of allowed *context switches*, *i.e.,* the number points along a computation where control passes from one thread to another [Qadeer and Rehof 2005]. (Note that a fixed context bound may only compute an *underapproximation* of the reachable states, and unsafe computations may be missed.) The proof is a combination of the computability of the reachability relation for sequential procedural programs and reachability. An alternate elegant proof reduces context bounded reachability with k context switches to sequential program reachability over a larger state space [Lal et al. 2008]. In practice, context bounding provides a useful underapproximation to reachability, and empirical results suggest that a low value of the context bound is already sufficient to discover bugs [Qadeer and Wu 2004; Musuvathi and Qadeer 2007]. Recently, symbolic implementations have been suggested for context bounded reachability [Suwimonteerabuth et al. 2008].

8. HEAP DATA STRUCTURES

♣ TO BE UPDATED♣

So far, we have assumed a simple program model where we have ignored the effect of heap data structures. Heap data structures represent one of the biggest challenges to software model checkers. The challenge of automatically reasoning about data structures stems from the need to reason about relationships between the unbounded number of values comprising the structure. Thus, the verification tool requires some way to generalize relationships over specific values into universally quantified facts that hold over the structure, and dually, to instantiate quantified facts to obtain relationships over particular values.

The class of imp+heap extends the class of imp by having a global and unbounded *memory* array, and operations *read* and *write* to read the memory at an index or to write a value (which could be another index) to an index. The reads and writes on the array are governed by McCarthy's axioms:

$$read(write(memory, index, value), index') = read(memory, index')$$

if $index \neq index'$ and

$$read(write(memory, index, value), index') = value$$

if $index = index'$. For a language like C, the analysis of programs in the presence of the heap become problematic, since two syntactically distinct expressions e_1 and e_2 could refer to the same memory location, and hence updating the memory by writing to location e_1 require updating information about the contents of the (syntactically different) location e_2.

We now briefly discuss some of the approaches that have been proposed. Our coverage of this topic is deliberately brief, since the vast literature on shape analysis requires its separate survey, and since the scalable model checking of heap-based data structures is perhaps the least understood direction in software model checking.

Alias analysis determines whether two pointers refer to the same heap cell [Muchnick 1997]. There is a wide spectrum of alias analyses that span the precision-scalability spectrum [Hind 2001; Hardekopf and Lin 2007]. Software model checkers like SLAM and BLAST use a precomputed alias analysis to determine whether an assigment $*\mathtt{x} \leftarrow e$ can affect the value of $*\mathtt{y}$. The SATURN verifier uses a combination of predicate abstraction, bounded model checking and procedure summarization to computes a precise path- and context- sensitive pointer analysis [Hackett and Aiken 2006] at the same time as the rest of the analysis. However, alias analysis is only useful for reasoning about explicitly named heap cells, but not unbounded data structures.

Shape analysis attempts to characterize collections of heap cells reachable from particular pointers, e.g. to determine whether the cells form a list or a tree and so on [Chase et al. 1990; Ghiya and Hendren 1996]. Early shape analyses used dataflow analyses over specialized lattices designed to capture properties of particular structures. [Sagiv et al. 2002] shows how three-valued logic could be used as a foundation for a parameterized framework for designing shape analyses. The framework is instantiated by supplying predicates that capture different relationships between cells (e.g. that one cell points to another, one cell is reachable from another), and by supplying the functions that determine how the predicates are updated by particular assignments. The tool TVLA [Lev-Ami and Sagiv 2000] implements these ideas and has been used to verify non-trivial data structure properties. Similar ideas were used to build a model checker capable of verifying concurrent, heap-manipulating programs [Yahav 2001]. Finally, [Gopan et al. 2005] shows how three-valued predicates can be combined with a numerical domain to verify properties of array manipulating programs.

Separation Logic [Reynolds 2002] was designed to enable modular reasoning about heap-manipulating programs. It extends classical Hoare logic [Hoare 1969] with two logical operators *separating conjunction* (typically written as $*$) and *separating implication* (typically written as $-\!\!*$), which are used to construct assertions over *disjoint* parts of the heap. For example, an assertion of the form $A*B$ says that there is one set of heap cells that satisfy A and a disjoint set of cells that satisfy B. The key advantage of this logic is that it allows one to crisply specify which parts of the heap are touched by a given piece of code, and allows one to compose properties of sub-heaps to get properties of the entire heap. While the logic was originally designed as a calculus for manually verifying low-level pointer manipulating programs, it has subsequently become the basis for several abstract interpretation based verifiers. To do so, the analysis designer specifies: (1) represents shape properties using recursively defined predicates, where the predicates are defined using the separating operators, and, (2) a mechanism to generalize (*i.e.,* fold) and instantiate (*i.e.,* unfold) such predicates [Distefano et al. 2006; Magill et al. 2007; Yang et al. 2008]. A variant of this approach is to extract the abstract domain

from programmer-specified (recursive) checker definitions [Chang and Rival 2008], and shows how the resulting shape domain can be combined with other abstractions to synthesize richer invariants like sortedness.

Reachability Predicates and logics built around them [Nelson 1983] provide another way to reason about heaps. The challenge is to design a *decidable* logic that is expressive enough to capture interesting invariants. The HAVOC tool uses an expressive logic with decidability and completeness guarantees [Lahiri and Qadeer 2008], but requires the user to provide loop invariants. [Gulwani et al. 2008] shows how to combine different abstract domains to obtain universally quantified domains that can capture properties of linked lists. [McMillan 2008] describes a technique that uses Craig interpolation to find universally quantified invariants for link lists.

9. LIVENESS AND TERMINATION

So far, we have focused on safety properties. We now focus on *liveness* properties which state, informally, that something good eventually happens. For finite state programs, and liveness properties specified in a temporal logic such as LTL [Emerson 1990], there is an automata-theoretic algorithm to check if the program satisfies the algorithm [Vardi and Wolper 1986]. Briefly, the algorithm constructs a *Büchi automaton* from the negation of the LTL property, and checks that the intersection of language of program behaviors and the language of the Büchi automaton is empty [Vardi and Wolper 1986; Vardi 1995]. Emptiness of the intersection can be checked by performing a nested depth-first search, looking for accepting cycles in the automaton [Courcoubetis et al. 1992]. This algorithm is implemented in the Spin model checker. A symbolic version of the algorithm (using BDD operations) is implemented in SMV. To verify arbitrary LTL properties of procedural programs, we need to track the contents of the control stack. [Bouajjani et al. 1994] shows how to precisely model check linear and branching time properties of *pushdown systems* by using automata to symbolically represent sets of stack configurations. [Esparza and Schwoon 2001] describes MOPED, which combines BDD-based symbolic representation for data, *i.e.,* program variables, with automata-based representation for stacks, in order to obtain an LTL model checking algorithm for Boolean programs. Finally, [Reps et al. 2005] describes *weighted pushdown systems*, a general analysis framework that encompasses interprocedural model checking of Boolean programs as well as a rich class of dataflow analysis problems.

Liveness properties are properties of infinite executions. Since execution-based model checkers, by definition, look at finite paths, a common assumption is to convert a liveness specification into a safety one, by bounding the number of steps within which "something good" happens. MACEMC is an execution-based model checker for distributed systems written in MACE, a domain-specific language built on top of C++. MACEMC has been used to find several complex liveness violations, for example, those in which a file does not get downloaded or the nodes of the system fail to organize into some topology, in a variety of distributed systems implementations. MACEMC combats state space explosion in two ways. First, rather than exploring the interleavings of low-level operations like network sends and receives, MACEMC exploits higher-level constructs in its modeling language MACE to explore only coarse-grained interleavings of event-transitions, each of which can

comprise of tens of low-level shared operations. Second, it combines exhaustive search with long random walks executed from the periphery of the exhaustive search in order to find bounded safety violations (*e.g.,* executions where a file is not downloaded in a certain number of steps), which indicate at least performance problems in code, if not violations of the liveness property.

9.1 Rank Functions

We now move to checking liveness properties for infinite-state systems. We focus on *termination*, a liveness property that requires that a program has no infinite computations. Formally, P is terminating if every computation $\langle \ell_0, s_0 \rangle \rightarrow \ldots \langle \ell_k, s_k \rangle$ reaches some state $\langle \ell_k, s_k \rangle$ which has no successor.

As we shall see, for many programs, termination can be proved only under certain assumptions, called *fairness requirements*, about the non-deterministic choices made during program execution. For example, a fairness requirement could enforce that a send over a non-deterministically lossy channel eventually must succeed. *Fair termination* is the property that a program terminates on all runs that satisfy the fairness requirements.

Just as safety properties can be reduced to reachability problems, liveness properties can be reduced to checking termination under fairness requirements [Vardi 1991]. For this reason, we shall concentrate in the rest of the section on techniques to prove fair termination for programs.

Just as state invariants formed the basis of safety proofs, *rank functions* form the basis for proofs of termination. The idea is to associate a rank with each intermediate state of computation, which decreases with every transition, and such that there is no infinitely decreasing sequence of ranks. We start with some definitions.

A partial order $\leq$ over a set A is *well-founded* iff there does not exist an infinite descending sequence of elements in A, *i.e.,* a sequence $a_0 > a_1 > \ldots$. A relation $R \subseteq A \times A$ is well-founded if there is no infinite sequence $a_0, a_1, \ldots$ such that for each $i \geq 0$ we have $a_i R a_{i+1}$. An element $a \in A$ is *minimal* if there is no $a' \in A$ with $a > a'$ (respectively, aRa'). For example, with the usual orderings, the natural numbers are well-founded, but the integers are not. From these definitions, it follows that a program is terminating iff the transition relation restricted to the reachable states is well-founded.

It is crucial to restrict the transition relation to reachable states: the transition relation of a terminating program may not by itself be well-founded, for example due to the presence of unreachable non-terminating loops. In general, finding the exact set of reachable transitions is not feasible, so we look for transition invariants. A *transition invariant* T is a superset of the transitive closure of the transition relation restricted to the reachable states. Techniques to prove termination reduce termination problems to check the well-foundedness of some transition invariant.

One way to compute a transition invariant is to add auxiliary variables to the set of program variables which track at each state the values of each variable in the previous state. This reduces computing transition invariants to state invariants and the techniques for computing state invariants apply.

Given a transition invariant, it may not be easy to check well-foundedness. In certain cases, for example, when the transition invariant can be expressed as linear constraints, one can algorithmically decide well-foundedness

[Podelski and Rybalchenko 2004a; Bradley et al. 2005].

Sometimes it can be hard to find a suitable single ranking function for a transition relation. In these cases, it is preferable to *compose* a termination argument out of simpler termination arguments. One way to do this is through the notion of disjunctive well-foundedness [Podelski and Rybalchenko 2004b]. A relation T is *disjunctively well-founded* if it is a finite union $T = T_1 \cup \ldots \cup T_k$ of well-founded relations. Every well-founded relation is (trivially) disjunctively well-founded, but not conversely. However, [Podelski and Rybalchenko 2004b] show that a relation R is well-founded if the transitive closure R^+ of R is contained in some disjunctively well-founded relation T. The relation T can be found as the union of transition invariants computed for parts of the program. This is the basis of TERMINATOR, a tool for proving termination of C programs [Cook et al. 2006].

Fairness and Liveness. Often, requiring termination for all possible behaviors is too strong. For example, the programmer may model certain aspects through non-deterministic choice, with an implicit assumption that such choices are fair. For example, one can model a scheduler as non-deterministically providing a resource to one or other process, with the assumption that both processes are picked infinitely often, or one can model asynchrony by modeling non-deterministic "stutter" steps, with the assumption that the process makes progress infinitely often. The standard way to rule out certain undesirable infinite behaviors from the scope of verification is through *fairness conditions* [Francez 1986]. Typically, a fairness condition can be translated to an automaton on infinite words [Vardi 1995]. The techniques for proving termination generalize to fair termination by taking a product of the program with the automaton, and checking for well-foundedness only for final states of the automaton. In fact, techniques based on transition invariants can directly reason about fairness [Pnueli et al. 2005].

Non-termination. In theory, there exists a rank function for every terminating program, however, this rank function may not be easy to find. Thus, given a program (and a transition invariant), a rank function is a concrete evidence of termination, but the inability to find a rank function does not immediately signify that the program has an infinite execution. In case an attempt at a proof of termination fails, this suggests one should apply a method specialized for proving *non-termination* to check if the "counterexample" is genuine. Non-termination has received relatively lesser attention in program verification. One way is through finding *recurrence sets* [Gupta et al. 2008]. A set of states R is recurrent if for every state $s \in R$, there is some successor s' of s such that $s' \in R$. It can be shown that a transition relation is not well-founded iff there is some recurrent set, and a program is non-terminating if one can find a recurrent set R which intersects the set of reachable states.

10. MODEL CHECKING AND SOFTWARE QUALITY

In the preceding sections we have traced the main developments in software model checking. We now indicate some recent interactions and synergies between model checking and allied fields.

10.1 Model Checking and Testing

Testing involves running software on a set of inputs. Testing is the primary technique to ensure software quality in the software industry. Sometimes the term *dynamic analysis* is used for testing, denoting that the program is actually executed and its outputs observed, as opposed to *static analysis* in which a *mathematical model* of the system is analyzed (e.g., in abstract model checking). Like most classifications, the boundaries are somewhat blurred, and some tools mix dynamic and static analysis. For example, Spin allows users to write arbitrary C code for part of the model that is executed during the model checking search.

Systematic Exploration. Testing and model checking are closely related: model checking is the exhaustive testing of program behaviors relative to a given property. For example, tools like Verisoft or MaceMC systematically test programs for all possible non-deterministic choices, and a sequence of values representing the outcome of non-deterministic choices can be seen as the "test input". For a large software system, the possible number of behaviors is so large that exhaustive testing is unlikely to finish within the software development budget. Hence, the goal of testing is to explore a subset of program behaviors that are "most likely" to uncover problems in the code, and a model checker can be used in "bug-finding" mode in which it searches as many behaviors as allowed within system resources. In this setting, optimizations derived for model checking, such as partial order reduction, are transferrable mutatis mutandis into optimizations of the testing process to rule out tests "similar" (in a precise sense) to the tests already run. Further, state space exploration tools can be configured to provide systematic *underapproximations* to program behaviors by fixing parameters such as input domain, search depth, or the number of context switches in concurrent code, and then checking behaviors within this underapproximation exhaustively. The latter approximation, called *context bounded analysis* [Qadeer and Rehof 2005], has been experimentally demonstrated to be effective for bug-finding for multi-threaded code even when the number of context switches is fixed to a small constant. Context bounded analysis has been implemented in CHESS, an execution-based enumerative model checker for multithreaded programs [Musuvathi and Qadeer 2007].

In *runtime verification*, the sequence of inputs and outputs of a running program are observed to ensure it satisfies a temporal requirement. The program may be terminated if the temporal requirements are observed to be violated [Kim et al. 2001; Havelund and Rosu 2004a; Havelund and Rosu 2004b].

Test Generation by Symbolic Evaluation. In a second direction, model checking, in combination with symbolic execution, can be used for test case generation. For example, in [Beyer et al. 2004; Xia et al. 2005] a software model checker is used to find program paths satisfying certain coverage conditions (e.g., a path reaching a particular location, or taking a particular branch), and the symbolic constraints generated from the path are solved by a constraint solver to produce test inputs. While the idea of using symbolic execution for test case generation is old [Clarke 1976], the combination with model checking allows search for test inputs satisfying particular coverage requirements to benefit from the search strategy of the model checker over an abstracted state space.

More recently, combined concrete and symbolic execution (or *concolic execu-*

tion) [Godefroid et al. 2005; Sen et al. 2005; Cadar et al. 2006] has been suggested as a strategy to combine symbolic execution and testing. In this technique, first proposed in [Godefroid et al. 2005] and independently in [Cadar et al. 2006], the program is run on concrete inputs (chosen, *e.g.,* at random) and while it is running, symbolic constraints on the execution path (as a function of symbolic inputs) are generated (e.g., through program instrumentation). By systematically negating symbolic constraints at conditional branches, and solving these constraints using a decision procedure, one generates a set of test inputs exploring every program path. The dynamic execution ensures that (a) parts of the code which do not depend on symbolic inputs are not tracked symbolically (this has been a major performance bottleneck of "pure" symbolic execution), and (b) program semantics hard to capture statically, e.g., dynamic memory allocation, can be tracked at run time. Additionally, run time values can be used to simplify symbolic constraints (at the cost of missing certain executions). For example, consider the program

```
x = input();
for (i = 0; i < 1000; i++) a[i] = 0;
```

Pure symbolic execution techniques would symbolically unroll the loop 1000 times, and the resulting constraints would slow down a decision procedure. In contrast, dynamic symbolic execution runs the loop concretely without generating additional symbolic constraints.

In [Gulavani et al. 2006; Beckman et al. 2008], the two ideas of CEGAR-based software model checking and dynamic symbolic execution have been combined to simultaneously search an abstract state space to produce a proof of correctness using predicate abstraction and to search for a failing test case using dynamic symbolic execution on counterexamples returned by the model checker. If dynamic symbolic execution finds the counterexample to be infeasible, usual counterexample analysis techniques can be used to refine the abstraction. The algorithm combines the abilities of CEGAR, to quickly explore abstract state spaces, and of dynamic symbolic execution, to quickly explore particular program paths for feasibility.

Dynamic detection of likely program invariants, where program invariants are guessed from dynamic runs [Ernst et al. 2001], demonstrates a second synergy between static analysis and dynamic analysis. The idea is to fix a language of potential invariants (e.g., a list of predicates on program variables) run the program on set of test inputs to see at each program point which of the potential invariants hold for each test run. Clearly, if a predicate fails to hold at a program point for a test run, it cannot be an invariant. Conversely though, even if a predicate holds at a program point for every test run, it may not be an invariant (unless the set of tests is exhaustive). In a second step, the set of potential invariants is used in a software verification tool to check both if the guesses are actual invariants and if the invariants are strong enough to prove program assertions [Nimmer and Ernst 2001].

10.2 Model Checking and Type Systems

Type systems are the most pervasive of all software verification techniques. Historically, the goal of types has been to classify program entities with a view towards ensuring that only well defined operations are carried out at run-time. One can view a type system as a technique for computing very coarse invariants over program

variables and expressions. For example, the fact that `x` has type `int` is essentially the invariant that in every reachable state of the program, an integer value is held in `x`. Thus, as type systems provide a scalable technique for computing coarse-grained invariants. Typically, these invariants have been "flow-insensitive" meaning they are facts that hold at *every* program point, and hence, they cannot be used to verify richer temporal properties. However, several recent approaches relax this restriction, and open the way to applying types to verify richer safety properties.

Typestates. [Strom and Yemini 1986] extend types with a finite set of states corresponding to the different stages of the value's lifetime. For example, a file handle type can be in two states: `open` if the handle refers to an open file, and, `closed` if the handle refers to a closed file. The types of primitive operations like `open`() or `close`() are annotated with pre- and post-conditions describing the appropriate input and output typestates, and a dataflow (or type-and-effect) analysis is used to determine the typestates of different objects at different program points, and hence, verify temporal properties like a file should not be read or written after it has been closed. Examples of such analyses include [Foster et al. 2002] which captures typestates using the notion of flow-sensitive type qualifiers, [Fahndrich and DeLine 2004] which introduces the notion of a *typestate interpretation*, a mapping from typestates to predicates over an objects fields, and shows how to exploit this notion to verify object-oriented programs using inheritance, and, [Field et al. 2005] which describes complexity results for typestate verification. Each of these algorithms can be viewed as model checking the program over a fixed abstraction generated by the product of the control-flow graph and the typestates [Chen and Wagner 2002]. Consequently, the algorithms are more efficient than general purpose software model checkers, but less precise as they as they ignore branches and other path information. [Das et al. 2002] shows how a modicum of path information can be regained by combining typestates with symbolic execution, leading to significant improvements in precision.

Dependent Types. Dependent types [Martin-Löf 1984] provide a complementary approach for encoding arbitrary invariants inside the type system, by refining the types with predicates that describe sets of values. For example, the refined type:

$$\{\nu{:}\texttt{int} \mid 0 \leq \nu \wedge \nu < \texttt{n}\}\ \texttt{list}$$

describes a list of integers, where each integer, represented by ν is greater than 0 and less than the value of some program variable `n`. [Freeman and Pfenning 1991] introduced the idea of type refinements by showing how restrictions on the structure of algebraic datatypes could capture regular properties (e.g. that a list has an even or odd number of [Xi and Pfenning 1999] extends ML with dependent types over a constraint domain C. Type checking is shown to be decidable modulo the decidability of the domain, thus allowing the programmer to specify and verify pre-and post- conditions that can be expressed over C. In essence, this result showed how classical Hoare-style verification (with pre- and post-conditions) could be combined with types to modularly verify programs using higher-order functions, polymorphism and recursive types. [Nanevski et al. 2008] makes the connection more explicit by showing how dependent types enable Hoare-style verification of imperative programs with higher-order functions. The above approaches require

that programmers provide type annotations corresponding to pre- and post- conditions for all functions. [Rondon et al. 2008] shows how the machinery of software model checking (predicate abstraction to be precise) can be brought to bear on the problem of *inferring* dependent type annotations, in much the same way as model checking can be viewed as a mechanism for synthesizing invariants. As a result one can combine the complementary strengths of model checking (local path- and value- information) and type systems (higher-order functions, recursive data, polymorphism) to verify properties that are well-beyond the abilities of either technique in isolation.

11. CONCLUSIONS

Software model checkers and related algorithmic verification tools hold the potential to close the gap between the programmer's intent and the actual code. However, the current generation of software model checking tools work best only for finite-state protocol properties, and we are still far away from proving functional properties of complex software systems. There are many remaining problems, both in scaling current techniques to large programs, and in devising algorithmic analyses for modern software systems. For example, scaling verification techniques in the presence of expressive heap abstractions and concurrent interactions remain outstanding open problems.

Many modern programming language features, such as object-orientation and dynamic dispatch, abstract data types, higher-order control flow and continuations, etc. are skirted in current algorithms and tools, and we would like to see verification tools exploiting language-level features. Similarly, common practice in large-scale software engineering, such as design patterns, the use of information hiding and layering, incremental development with regression tests, and design and architectural information is not exploited by current tools, but could be crucial in scaling tools to large scale software projects. Each of these directions would form excellent research topics.

Despite the shortcomings, we believe software model checking has made excellent progress in the past decade by selecting winning combinations of ideas from many disciplines, and in several settings, verification techniques can complement or outperform more traditional quality assurance processes based on testing and code inspection in terms of cost and effectiveness.

On the whole, it is unlikely that just software model checking tools will turn software development into a routine effort. Developing reliable software is too complex a problem, and has social aspects in addition to technical ones. However, we believe that the emergence of automatic tools and their use in the development process will help amplify programmer productivity by checking for partial properties of code, leaving the programmer more time to focus on more complex issues.

Acknowledgments. We thank Jay Misra and Tony Hoare for encouraging us to write this survey and several useful comments along the way. We thank Michael Emmi, Aarti Gupta, Roman Manevich, Sriram Rajamani, Andrey Rybalchenko, and the anonymous referees for feedback on earlier drafts. This work was sponsored in part by the National Science Foundation grants CCF-0546170, CCF-0702743, and CNS-0720881.

REFERENCES

Abadi, M. and Lamport, L. 1990. Composing specifications. In *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds. LNCS 430. Springer-Verlag.

Agerwala, T. and Misra, J. 1978. Assertion graphs for verifying and synthesizing programs. Tech. Rep. 83, University of Texas, Austin.

Alur, R., Dang, T., and Ivancic, F. 2006. Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci. 354,* 2, 250–271.

Alur, R. and Henzinger, T. 1999. Reactive modules. *Formal Methods in System Design 15,* 1, 7–48.

Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., and Tasiran, S. 1998. Mocha: Modularity in model checking. In *CAV 98: Computer-Aided Verification*, A. Hu and M. Vardi, Eds. Lecture Notes in Computer Science 1427. Springer-Verlag, 521–525.

Alur, R., Itai, A., Kurshan, R., and Yannakakis, M. 1995. Timing verification by successive approximation. *Information and Computation 118,* 1, 142–157.

Apt, K. and Olderog, E.-R. 1991. *Verification of Sequential and Concurrent Programs*. Springer-Verlag.

Armando, A., Mantovani, J., and Platania, L. 2006. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking Software: SPIN Workshop*. Lecture Notes in Computer Science, vol. 3925. Springer, 146–162.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program. 72,* 1-2, 3–21.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. 2006. Thorough static analysis of device drivers. In *EuroSys*. 73–85.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *PLDI*. ACM, 203–213.

Ball, T., Millstein, T. D., and Rajamani, S. K. 2005. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst. 27,* 2, 314–343.

Ball, T., Podelski, A., and Rajamani, S. K. 2001. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*. LNCS 2031. Springer, 268–283.

Ball, T., Podelski, A., and Rajamani, S. K. 2002. Relative completeness of abstraction refinement for software model checking. In *TACAS*. Lecture Notes in Computer Science, vol. 2280. Springer, 158–172.

Ball, T. and Rajamani, S. 2002a. Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research.

Ball, T. and Rajamani, S. 2002b. The SLAM project: debugging system software via static analysis. In *POPL*. ACM.

Ball, T. and Rajamani, S. K. 2000a. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*. LNCS 1885. Springer, 113–130.

Ball, T. and Rajamani, S. K. 2000b. Boolean programs: a model and process for software analysis. Tech. Rep. MSR Technical Report 2000-14, Microsoft Research.

Beckman, N., Nori, A. V., Rajamani, S. K., and Simmons, R. J. 2008. Proofs from tests. In *ISSTA*. 3–14.

Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Generating tests from counterexamples. In *ICSE 04: Software Engineering*. 326–335.

Beyer, D., Henzinger, T., Majumdar, R., and Rybalchenko, A. 2007. Path invariants. In *PLDI*.

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. 2007. The software model checker blast. *STTT 9,* 5-6, 505–525.

Beyer, D., Henzinger, T. A., and Théoduloz, G. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV*. 504–518.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. 1999. Symbolic model checking using sat procedures instead of bdds. In *DAC*. 317–320.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., and Rival, X. 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*. Lecture Notes in Computer Science 2566. Springer-Verlag, 85–108.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., and Rival, X. 2003. A static analyzer for large safety-critical software. In *PLDI 03: Programming Languages Design and Implementation*. ACM, 196–207.

Bouajjani, A., Esparza, J., and Maler, O. 1994. Reachability analysis of pushdown automata: application to model checking. In *CONCUR 97: Concurrency Theory*. LNCS 1243. Springer-Verlag, 135–150.

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*. 62–73.

Bouajjani, A., Fernandez, J.-C., and Halbwachs, N. 1990. Minimal model generation. In *CAV 90: Computer-aided Verification*, R. Kurshan and E. Clarke, Eds. LNCS 531. Springer-Verlag, 197–203.

Bradley, A., Manna, Z., and Sipma, H. 2005. The polyranking principle. In *Proc. ICALP*. LNCS 3580. Springer, 1349–1361.

Brat, G., Drusinsky, D., Giannakopolou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Washington, R., and Visser, W. 2004. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in Systems Design 25*.

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., and Sebastiani, R. 2008. The MathSAT 4 SMT solver. In *CAV*. 299–303.

Bryant, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35,* 8, 677–691.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98,* 2, 142–170.

Bustan, D. and Grumberg, O. 2003. Simulation-based minimization. *ACM Transactions on Computational Logic 4*, 181–206.

Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D. 2006. EXE: automatically generating inputs of death. In *CCS*.

Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., and Yorav, K. 2004. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design 25,* 2-3, 129–166.

Chaki, S., Clarke, E., Kidd, N., Reps, T., and Touili, T. 2006. Verifying concurrent message-passing c programs with recursive calls. In *TACAS 06: Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 3920. Springer, 334–349.

Chaki, S., Clarke, E. M., Groce, A., and Strichman, O. 2003. Predicate abstraction with minimum predicates. In *CHARME*. 19–34.

Chandra, S., Godefroid, P., and Palm, C. 2002. Software model checking in practice: an industrial case study. In *ICSE*. 431–441.

Chandy, K. and Misra, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company.

Chang, B. E. and Rival, X. 2008. Relational inductive shape analysis. In *POPL*. 247–260.

Chase, D., Wegman, M., and Zadeck, F. 1990. Analysis of pointers and structures. In *PLDI 90: Programming Languages Design and Implementation*. ACM, 296–310.

Chaudhuri, S. 2008. Subcubic algorithms for recursive state machines. In *POPL*. 159–169.

Chen, H. and Wagner, D. 2002. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security 2002*. 235–244.

Clarke, E. M. and Emerson, E. A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*. LNCS 131. Springer, 52–71.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *CAV 00:*. LNCS 1855. Springer, 154–169.

Clarke, E. M., Grumberg, O., and Long, D. 1992. Model checking and abstraction. In *POPL 92: Principles of Programming Languages*. ACM, 343–354.

Clarke, L. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions in Software Engineering 2(2)*, 215–222.

Cook, B., Podelski, A., and Rybalchenko, A. 2006. Termination proofs for systems code. In *Proc. PLDI*. ACM, 415–426.

Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. 1992. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design 1*, 275–288.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *ISOP*. 106–130.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL 77*. ACM, 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *POPL*. 269–282.

Cousot, P. and Cousot, R. 2000. Temporal abstract interpretation. In *POPL 00: Principles of Programming Languages*. ACM, 12–25.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages*. ACM Press.

Craig, W. 1957. Linear reasoning. *J. Symbolic Logic 22*, 250–268.

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

Das, S., Dill, D. L., and Park, S. 1999. Experience with predicate abstraction. In *CAV 99: Computer-Aided Verification*. LNCS 1633. Springer, 160–171.

de Moura, L. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *TACAS*. 337–340.

de Moura, L., Ruess, H., and Sorea, M. 2002. Lazy theorem proving for bounded model checking over infinite domains. In *CADE 02: Automated Deduction*. Lecture Notes in Computer Science, vol. 2392. Springer, 438–455.

Dijkstra, E. 1976. *A Discipline of Programming*. Prentice-Hall.

Dill, D. 1996. The Murphi verification system. In *CAV 96: Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, 390–393.

Dillenburg, J. F. and Nelson, P. C. 1994. Perimeter search. *Artif. Intell. 65,* 1, 165–178.

Distefano, D., O'Hearn, P. W., and Yang, H. 2006. A local shape analysis based on separation logic. In *TACAS*. 287–302.

Dutertre, B. and Moura, L. D. Yices SMT solver. `http://yices.csl.sri.com/`.

Dwyer, M. and Clarke, L. 1994. Data flow analysis for verifying properties of concurrent programs. In *FSE 94: Foundations of Software Engineering*. ACM, 62–75.

Edelkamp, S., Leue, S., and Lluch-Lafuente, A. 2004. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer 5*, 247–267.

Een, N. and Sorensson, N. 2003. An extensible SAT solver. In *SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing*. LNCS 2919. Springer, 502–518.

Emerson, E. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier Science Publishers, 995–1072.

Emerson, E. and Sistla, A. 1996. Symmetry and model checking. *Formal Methods in System Design 9*, 105–131.

Ernst, M., Cockrell, J., Griswold, W. G., and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering 27(2)*, 1–25.

Esparza, J. and Schwoon, S. 2001. A BDD-based model checker for recursive programs. In *CAV*. 324–336.

Fahndrich, M. and DeLine, R. 2004. Typestates for objects. In *ECOOP 04: Object-Oriented Programming*. LNCS 3086. Springer, 465–490.

Field, J., Goyal, D., Ramalingam, G., and Yahav, E. 2005. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program. 58,* 1-2, 57–82.

Fischer, J., Jhala, R., and Majumdar, R. 2005. Joining dataflow with predicates. In *ESEC/FSE 2005: Foundations of Software Engineering*. ACM, 227–236.

Flanagan, C., Freund, S., and Qadeer, S. 2002. Thread-modular verification for shared-memory programs. In *ESOP 02: European Symposium on Programming*. LNCS 2305. Springer, 262–277.

Flanagan, C., Freund, S., Qadeer, S., and Seshia, S. 2005. Modular verification of multithreaded programs. *Theoretical Computer Science 338*, 153–183.

Flanagan, C., Joshi, R., and Leino, K. R. M. 2001. Annotation inference for modular checkers. *Information Processing Letters (to appear)*.

Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *PLDI*. 234–245.

Flanagan, C. and Qadeer, S. 2002. Predicate abstraction for software verification. In *POPL 02: Principles of Programming Languages*. ACM, 191–202.

Flanagan, C. and Saxe, J. 2000. Avoiding exponential explosion: generating compact verification conditions. In *POPL 00: Principles of Programming Languages*. ACM, 193–205.

Floyd, R. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*. American Mathematical Society, 19–32.

Foster, J., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*. ACM, 1–12.

Francez, N. 1986. *Fairness*. Springer-Verlag.

Fraser, R., Kamhi, G., Ziv, B., Vardi, M., and Fix, L. 2000. Prioritized traversal: efficient reachability analysis for verification and falsification. In *CAV 00: Computer-Aided Verification*. Lecture Notes in Computer Science 1855. Springer, 389–402.

Freeman, T. and Pfenning, F. 1991. Refinement types for ML. In *PLDI*.

Ghiya, R. and Hendren, L. J. 1996. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL*. 1–15.

Giannakopoulou, D. and Pasareanu, C. S. 2008. Special issue on learning techniques for compositional reasoning. *Formal Methods in System Design 32,* 3, 173–174.

Giesl, J. and Kapur, D. 2001. Decidable classes of inductive theorems. In *IJCAR 2001: International Joint Conference on Automated Reasoning*. Lecture Notes in Computer Science 2083. Springer, 469–484.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science 1032. Springer.

Godefroid, P. 1997. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*. ACM, 174–186.

Godefroid, P., Klarlund, N., and Sen, K. 2005. Dart: directed automated random testing. In *PLDI*. 213–223.

Gopan, D., Reps, T. W., and Sagiv, S. 2005. A framework for numeric analysis of array operations. In *POPL*. 338–350.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *CAV*. LNCS 1254. Springer, 72–83.

Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. 2008. Automatically refining abstract interpretations. In *TACAS*. 443–458.

Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., and Rajamani, S. K. 2006. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*. 117–127.

Gulwani, S., McCloskey, B., and Tiwari, A. 2008. Lifting abstract interpreters to quantified logical domains. In *POPL*. 235–246.

Gulwani, S. and Tiwari, A. 2006. Combining abstract interpreters. In *PLDI 2006: Programming Languages Design and Implementation*. ACM, To Appear.

Gupta, A., Henzinger, T., Majumdar, R., Rybalchenko, A., and Xu, R. 2008. Proving non-termination. In *POPL 08: Principles of Programming Languages*. ACM.

Hackett, B. and Aiken, A. 2006. How is aliasing used in systems software? In *SIGSOFT FSE*. 69–80.

Hardekopf, B. and Lin, C. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*. 290–299.

Hart, P., Nilsson, N., and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4 2*, 100–107.

Havelund, K. and Pressburger, T. 2000. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT) 2(4)*, 72–84.

Havelund, K. and Rosu, G. 2004a. Efficient monitoring of safety properties. *Software Tools for Technology Transfer 6*.

Havelund, K. and Rosu, G. 2004b. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design 24*.

Henzinger, T., Jhala, R., Majumdar, R., and McMillan, K. 2004. Abstractions from proofs. In *POPL 04*. ACM.

Henzinger, T., Jhala, R., Majumdar, R., and Qadeer, S. 2003. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*. Lecture Notes in Computer Science. Springer-Verlag.

Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *POPL 02: Principles of Programming Languages*. ACM, 58–70.

Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Race checking by context inference. In *PLDI 2004: Programming Languages Design and Implementation*. ACM, 1–12.

Hind, M. 2001. Pointer analysis: haven't we solved this problem yet? In *PASTE*. 54–61.

Hoare, C. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12*, 576–580.

Holzmann, G. 1997. The Spin model checker. *IEEE Transactions on Software Engineering 23,* 5 (May), 279–295.

Holzmann, G. J., Joshi, R., and Groce, A. 2008. Tackling large verification problems with the swarm tool. In *SPIN*. 134–143.

Ip, C. and Dill, D. 1996. Better verification through symmetry. *Formal Methods in System Design 9*, 41–75.

Ivancic, F., Yang, Z., Ganai, M. K., Gupta, A., and Ashar, P. 2008. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci. 404,* 3, 256–274.

Ivancic, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., and Ashar, P. 2005. F-soft: Software verification platform. In *CAV*. 301–306.

Jackson, D. 2006. *Software Abstractions: Logic, language, and analysis*. MIT Press.

Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., and Wang, C. 2006. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*. 137–151.

Jha, S. K., Krogh, B. H., Weimer, J. E., and Clarke, E. M. 2007. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HSCC*. 287–300.

Jhala, R. and Majumdar, R. 2005. Path slicing. In *PLDI 05: Programming Language Design and Implementation*. ACM, 38–47.

Jhala, R. and McMillan, K. 2006. A practical and complete approach to predicate refinement. In *TACAS 06*. LNCS 2987. Springer-Verlag, 298–312.

Jhala, R. and McMillan, K. L. 2005. Interpolant-based transition relation approximation. In *CAV 05*. LNCS. Springer, 39–51.

Kahlon, V. and Gupta, A. 2007. On the analysis of interacting pushdown systems. In *POPL*. 303–314.

Kim, M., Kannan, S., Lee, I., and Sokolsky, O. 2001. Java-MaC: a run-time assurance tool for Java. In *RV 01: Runtime Verification*. ENTCS 55. Elsevier.

Korf, R. 1985. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence 27*, 97–109.

Kroening, D., Clarke, E., and Yorav, K. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC 03: Design Automation Conference*. ACM, 368–371.

Kurshan, R. 1994. *Computer-aided Verification of Coordinating Processes*. Princeton University Press.

Lahiri, S. K. and Qadeer, S. 2008. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*.

Lal, A., Touili, T., Kidd, N., and Reps, T. W. 2008. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*. 282–298.

Larsen, K., Larsson, F., Pettersson, P., and Yi, W. 1997. Efficient verification of real-time systems: Compact data structure and state-space reductions. In *RTSS*. IEEE, 14–24.

Leino, K. R. M. and Nelson, G. 1998. An extended static checker for Modula-3. In *CC 98: Compiler Construction*. LNCS 1383. Springer, 302–305.

Lev-Ami, T. and Sagiv, S. 2000. TVLA: A system for implementing static analyses. In *SAS*. LNCS 1824. Springer, 280–301.

Lluch-Lafuente, A. 2003. Directed search for the verification of communication protocols. Ph.D. thesis.

Loiseauxc, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. 1995. Property-preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design 6*, 11–44.

Magill, S., Berdine, J., Clarke, E. M., and Cook, B. 2007. Arithmetic strengthening for shape analysis. In *SAS*. 419–436.

Manna, Z. and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

Martin-Löf, P. 1984. Constructive mathematics and computer programming. *Royal Society of London Philosophical Transactions Series A 312*, 501–518.

McMillan, K. 1993. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers.

McMillan, K. L. 2004. An interpolating theorem prover. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*. 16–30.

McMillan, K. L. 2006. Lazy abstraction with interpolants. In *CAV 2006*. LNCS. Springer-Verlag, 123–136.

McMillan, K. L. 2008. Quantified invariant generation using an interpolating saturation prover. In *TACAS*. 413–427.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *DAC 01: Design Automation Conference*. 530–535.

Muchnick, S. 1997. *Advanced Compiler Design and Implementation*. Morgan-Kaufman.

Musuvathi, M. and Engler, D. R. 2004. Model checking large network protocol implementations. In *NSDI*. 155–168.

Musuvathi, M. and Qadeer, S. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*. 446–455.

Namjoshi, K. S. and Kurshan, R. P. 2000. Syntactic program transformations for automatic abstraction. In *CAV*. 435–449.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. 2008. Ynot: Reasoning with the awkward squad. In *ICFP*.

Necula, G. C. and Lee, P. 2000. Proof generation in the Touchstone theorem prover. In *CADE 00: Computer-Aided Deduction*. LNCS 1831. Springer, 25–44.

Nelson, G. 1981. Techniques for program verification. Tech. Rep. CSL81-10, Xerox Palo Alto Research Center.

Nelson, G. 1983. Verifying reachability invariants of linked structures. In *POPL*. 38–47.

Nelson, G. and Oppen, D. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM 27,* 2, 356–364.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*. Springer-Verlag.

Nimmer, J. and Ernst, M. 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci. 55(2)*.

Penix, J., Visser, W., Park, S., Pasareanu, C., Engstrom, E., Larson, A., and Weininger, N. 2005. Verifying time partitioning in the DEOS scheduling kernel. *Formal Methods in Systems Design 26*.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 46–57.

Pnueli, A., Podelski, A., and Rybalchenko, A. 2005. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 3440. Springer, 124–139.

Podelski, A. and Rybalchenko, A. 2004a. A complete method for the synthesis of linear ranking functions. In *VMCAI*. 239–251.

Podelski, A. and Rybalchenko, A. 2004b. Transition invariants. In *LICS 04: Logic in Computer Science*. IEEE.

Qadeer, S., Rajamani, S. K., and Rehof, J. 2004. Summarizing procedures in concurrent programs. In *POPL*. 245–255.

Qadeer, S. and Rehof, J. 2005. Context-bounded model checking of concurrent software. In *TACAS*. 93–107.

Qadeer, S. and Wu, D. 2004. Kiss: keep it simple and sequential. In *PLDI*. 14–24.

Queille, J. and Sifakis, J. 1981. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds. LNCS 137. Springer-Verlag, 337–351.

Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22,* 2, 416–430.

Reps, T., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*. ACM, 49–61.

Reps, T. W., Schwoon, S., Jha, S., and Melski, D. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program. 58,* 1-2, 206–263.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74.

Rondon, P., Kawaguchi, M., and Jhala, R. 2008. Liquid types. In *PLDI*. 158–169.

Russell, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall.

Rybalchenko, A. and Sofronie-Stokkermans, V. 2007. Constraint solving for interpolation. In *VMCAI*. 346–362.

Sagiv, S., Reps, T. W., and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24,* 3, 217–298.

Saidi, H. 2000. Model checking guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*. LNCS 1824, Springer, 377–396.

Saïdi, H. and Shankar, N. 1999. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*. LNCS 1633. Springer, 443–454.

Sankaranarayanan, S., Sipma, H. B., and Manna, Z. 2005. Scalable analysis of linear systems using mathematical programming. In *VMCAI*. 25–41.

Schmidt, D. 1998. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*. ACM, 38–48.

Sen, K., Marinov, D., and Agha, G. 2005. CUTE: a concolic unit testing engine for C. In *FSE*.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 189–233.

Shostak, R. 1984. Deciding combinations of theories. *Journal of the ACM 31,* 1, 1–12.

Silva, J. P. M. and Sakallah, K. A. 1996. Grasp - a new search algorithm for satisfiability. In *ICCAD*. 220–227.

Sistla, A., Gyuris, V., and Emerson, E. 2000. SMC: A symmetry-based model checking. *ACM Transactions on Software Engineering Methodology 9*, 133–166.

Somenzi, F. 1998. Colorado University decision diagram package. `http://vlsi.colorado.edu/pub/`.

Stern, U. and Dill, D. L. 1998. Using magnetic disk instead of main memory in the mur*hi* verifier. In *CAV*. 172–183.

Strom, R. and Yemini, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12(1)*, 157–171.

Suwimonteerabuth, D., Esparza, J., and Schwoon, S. 2008. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN*. 270–287.

Turing, A. M. 1936. On computable numbers, with an application to the eintscheidungsproblem. In *Proceedings of the London Mathematical Soceity*. 230–265.

Vardi, M. 1991. Verification of concurrent programs — the automata-theoretic framework. *Annals of Pure and Applied Logic 51*, 79–98.

Vardi, M. 1995. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop Proceedings)*. Lecture Notes in Computer Science, vol. 1043. Springer, 238–266.

Vardi, M. and Wolper, P. 1986. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences 32*, 183–221.

Vardi, M. and Wolper, P. 1994. Reasoning about infinite computations. *Information and Computation 115,* 1, 1–37.

Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. 2003. Model checking programs. *Automated Software Engineering Journal 10*.

Wang, C., Yang, Z., Gupta, A., and Ivancic, F. 2007. Using counterexamples for improving the precision of reachability computation with polyhedra. In *CAV*. 352–365.

Xi, H. and Pfenning, F. 1999. Dependent types in practical programming. In *POPL*. 214–227.

Xia, S., Vito, B. D., and Muñoz, C. 2005. Automated test generation for engineering applications. In *ASE 05: Automated Software Engineering*. ACM, 283–286.

Xie, Y. and Aiken, A. A. 2005. Scalable error detection using boolean satisfiability. In *POPL 05: Principles of Programming Languages*. ACM.

Yahav, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL 01: Principles of Programming Languages*. ACM Press, 27–40.

Yang, C. H. and Dill, D. L. 1998. Validation with guided search of the state space. In *DAC*. 599–604.

Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. W. 2008. Scalable shape analysis for systems code. In *CAV*. 385–398.

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. 2004. Using model checking to find serious file system errors. In *OSDI 04: Operating System Design and Implementation*. Usenix Association.

Yang, Z., Wang, C., Gupta, A., and Ivancic, F. 2006. Mixed symbolic representations for model checking software programs. In *MEMOCODE*. 17–26.

Yannakakis, M. 1990. Graph theoretic methods in database theory. In *Proc. 9th ACM Symp. on Principles of Database Systems*. ACM Press, 203–242.