A Survey of Automated Techniques for Formal Software Verification

Vijay D'Silva Daniel Kroening Georg Weissenbacher

***Abstract*—The quality and the correctness of software is often the greatest concern in electronic systems. Formal verification tools can provide a guarantee that a design is free of specific flaws. This paper surveys algorithms that perform automatic, static analysis of software to detect programming errors or prove their absence. The three techniques considered are static analysis with abstract domains, model checking, and bounded model checking. A short tutorial on these techniques is provided, highlighting their differences when applied to practical problems. The paper also surveys the tools that are available implementing these techniques, and describes their merits and shortcomings.**

***Index Terms*—Model Checking, Software Verification, Static Analysis, Predicate Abstraction, Bounded Model Checking**

I. Introduction

Correctness of computer systems is critical in today's information society. Though modern computer systems consist of complex hardware and software components, ensuring the correctness of the software part is often a greater problem than that of the underlying hardware. This is especially true of software controlling transportation and communication infrastructure. The website [52] documents this problem by providing over one-hundred "software horror stories".

Manual inspection of complex software is error-prone and costly, so tool support is in dire need. Several tools attempt to uncover design flaws using test vectors to examine specific executions of a software system. Formal verification tools, on the other hand, can check the behavior of a design for *all* input vectors. Numerous formal tools to hunt down functional design flaws in silicon are available and in wide-spread use. In contrast, the market for tools that address software quality is still in its infancy.

Research in software quality has an enormous breadth. We focus on methods satisfying two criteria:

1) The method provides a rigorous *guarantee* of quality.
2) The method should be highly *automated* and scalable to cope with the enormous complexity of software systems.

In practice, quality guarantees rarely refer to 'total correctness' of a design, as ensuring the absence of all bugs in a design is usually too expensive. However, a guarantee that specific flaws are absent is achievable and is a good metric of quality. Due to the first criterion, we do not survey random testing and automated test-case generation, though metrics such as branch coverage can be viewed as quality guarantees.

The second criterion, automation, implies that we exclude techniques such as dynamic verification using test vectors, and in particular, unit testing. We also exclude formal techniques requiring substantial manual effort, e.g., tools that require user interaction to construct a proof (e.g., ACL2, PVS or HOL), techniques that use refinement steps to construct programs from specifications (e.g., the B-method [1]) and tools that require significant annotations from the programmer (e.g., ESC/Java [61]). Consequently, we only survey static analysis techniques, which do not rely on program execution and require minimal user interaction.

Outline

The three techniques we survey are *Abstract Static Analysis*, *Model Checking*, and *Bounded Model Checking*. We begin with abstract static analysis (§ II). Light-weight versions of these techniques are used in software development tools, e.g., for pointer analysis in modern compilers. A formal basis for such techniques is *Abstract Interpretation*, introduced by Cousot and Cousot [46]. We briefly describe the formal background and then summarize tools that implement them.

The second part of the survey addresses *Model Checking* for software (§ III). Model Checking was introduced by Clarke and Emerson [36], and independently by Queille and Sifakis [97]. The basic idea is to determine if a correctness property holds by exhaustively exploring the reachable states of a program. If the property does not hold, the model checking algorithm generates a *counterexample*, an execution trace leading to a state in which the property is violated. The main benefit of model checking compared to the techniques in § II is its ability to generate counterexamples. As the state space of software programs is typically too large to be analyzed directly, model checking is often combined with abstraction techniques. We survey *predicate abstraction*, a popular abstraction technique for software model checking.

The third part of the survey is dedicated to a formal technique that performs a depth-bounded exploration of the state space (§ IV). As this technique is a form of Model Checking, it is called *Bounded Model Checking* (BMC). We present BMC separately, as the requirement to analyze the entire state space is relaxed: BMC explores program behavior exhaustively, but only up to a given depth. Bugs that require longer paths are missed.

This research is supported by an award from IBM Research, by a Microsoft Graduate Scholarship, and by the European Union as part of the FP7-STREP MOGENTES.

V. D'Silva is with the Computer Science Department at ETH Zurich, Switzerland. E-mail: vijay.dsilva@inf.ethz.ch

D. Kroening is with the Computing Laboratory at Oxford University, UK. E-mail: kroening@comlab.ox.ac.uk

G. Weissenbacher is with the Computer Science Department at ETH Zurich, Switzerland. E-mail: georg.weissenbacher@inf.ethz.ch

II. Abstract Static Analysis

Static analysis encompasses a family of techniques for automatically computing information about the behavior of a program without executing it. Though such techniques are used extensively in compiler optimization, we focus on their use in program verification. Most questions about the behavior of a program are either undecidable or it is infeasible to compute an answer. Thus, the essence of static analysis is to *efficiently* compute approximate but *sound* guarantees: guarantees that are not misleading.

The notion of approximation is qualitative, as opposed to that used in most engineering problems. A sound approximation is one that can be relied upon. For example, a procedure for detecting division-by-zero errors in a program must consider all possible run-time values of variables involved in division operations. An approximate procedure may compute either a subset or a superset of these values and expressions. If a subset is computed and no error is found, returning '`No Division Errors`' is unsound because an error may still exist. If a superset is computed and no errors are found, returning '`No Division Errors`' is justified. A *spurious warning* is an error message about a bug that does not exist in the program. A *missed bug* is one that exists but is not reported by an analysis procedure. Due to the undecidability of static analysis problems, devising a procedure that does not produce spurious warnings and does not miss bugs is not possible.

We introduce the vocabulary of static analysis in § II-A and illustrate an abstract analysis in § II-B. We survey abstractions of numeric data types in § II-C and of the heap in § II-D. We conclude this section with a review and discussion of static analysis tools (§ II-E,§ II-F). The relationship between static analysis and model checking is briefly discussed in § III.

A. General Schema and Terminology

Static analysis techniques typically propagate a set of values through a program until the set *saturates*, i.e., does not change with further propagation. Mathematically, such analyses are modeled as iterative application of a monotone function. Saturation occurs when a fixed point of the function is reached. We illustrate this idea with a deliberately simplified example.

Example 1: We want to determine all values the variable `i` may take in the program below. Such information has various applications. If `i` is used as an array index, the analysis is useful for ensuring that the array bounds are not exceeded.

```
int i = 0;
do {
   assert(i <= 10);
   i = i+2;
} while (i < 5);
```

The control flow graph (CFG) for this program is shown in Figure 1(a). Each node is annotated with its *value-set,* i.e., the values `i` may have on reaching that program location. Let **Int** be the set of values a variable of type **int** can have. For the analysis, we propagate a value-set of `i` along the edges of the CFG adding new values as they are encountered. The annotations computed in the first two iterations are shown in grey and black, respectively, in Figure 1(a).

The initial value-set of `i`, being undefined, is **Int**. After the assignment `i = 0`, the value-set is $\{0\}$. The statement `assert(i <= 10)` does not change `i`, so $L3$ has the same annotation as $L2$. After the statement `i = i + 2`, the set $\{0\}$ is changed to $\{2\}$. As $2<5$, this set is propagated from $L4$ to $L2$. The set $\{0\}$ is propagated from $L1$ to $L2$ and $\{2\}$ is propagated from $L4$ to $L2$, so $L2$ is annotated with the union of $\{0\}$ and $\{2\}$. A node like $L2$ with multiple incoming edges is a *join point* and is labeled with the set obtained by merging all sets propagated to it. The value-set is repeatedly propagated along the CFG in Figure 1(a) until a fixed point is reached. □

The analysis above is a *concrete interpretation* of the program. Subsets of **Int** represent precise values of `i` during execution and constitute a *concrete domain.* Concrete interpretation is impractical because value-sets grow large rapidly and such naïve propagation along a CFG does not scale. In practice, precision is often traded for efficiency by using an approximation of the value-set, or by ignoring certain aspects of the program's structure during propagation. With regard to the latter approach, we use the following taxonomy of program analyses: a program analysis method is

- *Flow sensitive* if the order of execution of statements in the program is considered.
- *Path sensitive* if it distinguishes between paths through a program and attempts to consider only feasible ones.
- *Context sensitive* if method calls are analyzed differently based on the call site.
- *Inter-procedural* if the body of the method is analysed in the context of each respective call site.

If one of these conditions is ignored, the analysis technique is flow, path or context insensitive, respectively. An analysis not distinguishing the call sites of the methods is *intra-procedural.*

B. Abstract Interpretation

Peter Naur observed when working on the Algol compiler that abstract values may suffice for program analysis [95]. In early tools, the relation between an abstract analysis and a program's run-time behavior was often unclear. Cousot and Cousot [46] introduced abstract interpretation as a framework for relating abstract analyses to program execution.

An *abstract domain* is an approximate representation of sets of concrete values. An abstraction function is used to map concrete values to abstract ones. An *abstract interpretation* involves evaluating the behavior of a program on an abstract domain to obtain an approximate solution. An abstract interpretation can be derived from a concrete interpretation by defining counterparts of concrete operations, such as addition or union, in the abstract domain. If certain mathematical constraints between the abstract and concrete domains are met, fixed points computed in an abstract domain are guaranteed to be sound approximations of concrete fixed points [47].

Example 2: A possible abstract domain for the analysis in Example 1 is the set of intervals, $\{[a, b] | a \leq b\}$ where a and b are elements of **Int**. The concrete operations in Example 1

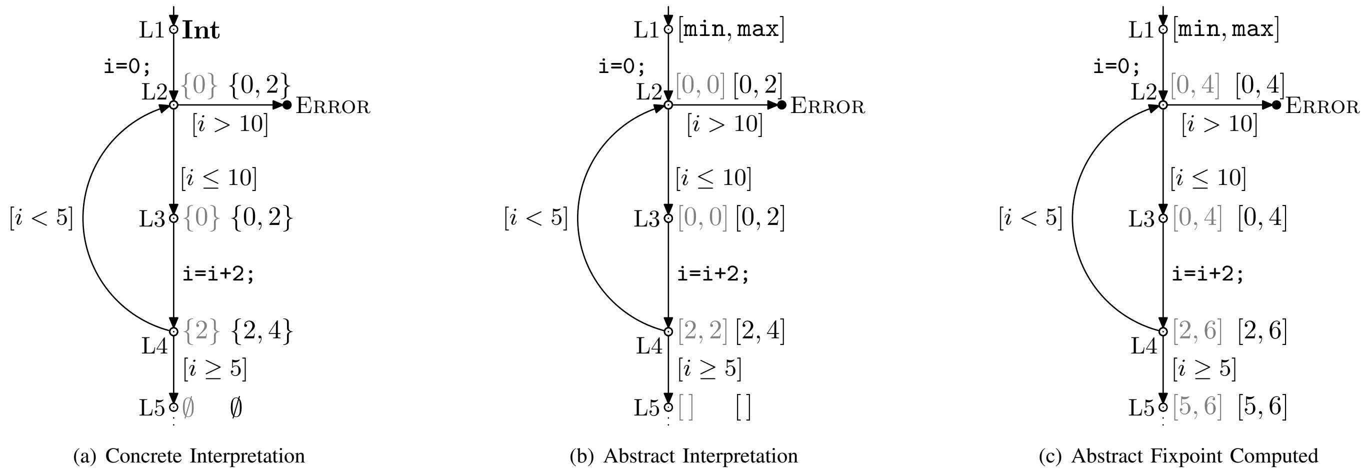


Fig. 1. Concrete and abstract interpretations of a program. The values of i are collected at program locations in (a). Data is abstracted using intervals in (b) and the fixed point is reached in (c). The labels on the edges denote the condition that has to hold for flow of control to pass the edge; the labels on the nodes correspond to variable valuations. The annotation of control locations computed in two successive iterations are shown in grey and black, respectively.

are addition and union on subsets of **Int**. Addition and union for intervals are defined in the intuitive manner.

The abstract interpretation of the program is illustrated in Figure 1(b). Let min and max be the minimum and maximum values of an **int** variable. The abstract value of i at $L1$ is $[\mathtt{min}, \mathtt{max}]$, and at $L2$ is $[0, 0]$. This interval is propagated to $L3$, and then transformed by $\mathtt{i} = \mathtt{i} + 2$ to $[2, 2]$. The union of the intervals $[0, 0]$ and $[2, 2]$ at the join point $L2$ is $[0, 2]$ as shown. The information that i was not 1 is lost by merging the intervals. The intervals computed in the second iteration are shown in black in Figure 1(b). Propagation ends when a fixed point is reached as shown in Figure 1(c). □

A typical choice of an abstract domain and sensitivity of the analysis yields imprecise but sound answers to program analysis problems. However, the abstract analysis may still require an infeasible number of iterations to reach a fixed point. A *widening operator* is used to accelerate and ensure termination of fixed point computations, usually by incurring further loss of precision. A complementary *narrowing operator* is then used to improve the precision of the solution.

C. Numerical Abstract Domains

Over the years, various abstract domains have been designed, particularly for computing invariants about numeric variables. The class of invariants that can be computed, and hence the properties that can be proved, varies with the expressive power of a domain. *Relational domains* can capture relationships between variables and *non-relational domains* cannot. An abstract domain is more precise than another if less information is lost. The information loss between different domains may be incomparable as we illustrate shortly. We now review common abstract domains beginning with simple non-relational ones and progressing to relational domains.

The domain of *Signs* has three values: $\{\mathtt{Pos}, \mathtt{Neg}, \mathtt{Zero}\}$. Intervals are more expressive as the values in the Signs domain are modeled by the intervals $[\mathtt{min}, 0]$, $[0, 0]$ and $[0, \mathtt{max}]$. The domain of *Parities* abstracts values as Even and Odd. Though the domain Parities has fewer elements than the domains of Signs or Intervals, it cannot be compared with the latter two. No interval can be used to model all odd or even values. The domain of *Congruences* generalizes Parities, representing a value v by $(v \bmod k)$ for some fixed integer k.

Consider the expression $1/(x - y)$. If abstract interpretation is used to show that $(x \bmod k) \neq (y \bmod k)$, we may conclude that division by zero does not occur in evaluating this expression. Congruences can be used to prove dis-equalities but not inequalities between variables. Non-relational domains are efficient to represent and manipulate. Unfortunately, even simple properties like $x \leq y$ cannot be represented.

We now survey some relational domains. *Difference Bound Matrices* (DBMs) are conjunctions of inequalities of the form $x - y \leq c$ and $\pm x \leq c$ [19]. They were initially used to analyze timed Petri-nets and later for model checking real-time systems [91], [110]. DBMs allow for more precise abstraction than intervals but do not admit constraints of the form $(-x - y \leq c)$. *Octagons* are a more expressive domain with equations of the form $(ax + by \leq c)$ where $a, b \in \{-1, 0, 1\}$ and c is an integer [92]. *Octahedra* [33] generalize Octagons to more than two variables. The domain of *Polyhedra* is among the earliest and most popular relational domains used to prove numerical properties of programs, and timing behavior of embedded software [48], [67]. A polyhedron is a conjunction of inequalities of the form $a_1x_1 + \cdots + a_nx_n \leq c$, where a_i and c are integers. Manipulating polyhedra involves computing convex hulls, which is computationally expensive. The time and space requirements of procedures for manipulating polyhedra are usually exponential in the number of variables.

The domains of Signs, Intervals, DBMs, Octagons, Octahedra and Polyhedra form a hierarchy in terms of expressive power. They are useful for computing inequality constraints about integer variables and for deriving such invariants for nested program loops. The related *ellipsoid domain* encodes non-linear relationships of the form $ax^2 + bxy + cy^2 \leq n$ and has been used to analyze real-time digital filters [59].

These relational domains cannot be used to prove dis-equalities. Consider the expression $1/(2 * x + 1 - y)$. An abstract domain for proving that division by zero does not occur should allow us to conclude that $2*x+1 \neq y$. The domain of *Linear Congruences* combines features of Polyhedra and Congruences, containing equations of the form $ax+by = c \bmod k$. If we can conclude that $2*x+1 \bmod k \neq y \bmod k$, the result of the expression above is well defined. In general, new domains can be designed from existing ones as required [47].

The domains described above have been used to prove properties of avionic and embedded software [25]. Numeric domains have also been used to analyze pointer behavior [54], string manipulations [55] and program termination [41].

D. Shape Analysis

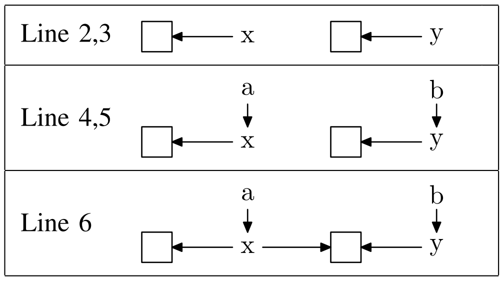
The next class of abstract analyses and domains we consider concern the heap and are essential for analyzing programs with pointers. *Alias analysis* is the problem of checking if two pointer variables access the same memory location. *Points-to analysis* requires identifying the memory locations a pointer may access [71]. *Shape analysis*, formulated by Reynolds [100] and later independently by Jones and Muchnick [81], generalizes these problems to that of verifying properties of dynamically created data structures, so called because such properties are related to the "shape" of the heap.

The abstract domain for points-to analysis contains *storage shape graphs* [32], also called *alias graphs* [72]. Nodes in the graph represent variables and memory locations and edges encode *points-to* relationships. Considering program structure significantly affects the efficiency of shape analysis. The fastest and most popular points-to analysis algorithms are flow-insensitive [4], [103], disregarding control flow. We illustrate a flow-insensitive analysis in the next example.

Example 3: Consider the program fragment below and the question "*Do* x *and* y *point to the same memory location?*"

```
1: int **a, **b, *x, *y;
2: x = (int*) malloc(sizeof(int));
3: y = (int*) malloc(sizeof(int));
4: a = &x;
5: b = &y;
6: *a = y;
```

The construction of a points-to graph is shown below.



The effect of malloc statements (lines 2,3) is modeled by adding a memory node and a pointer to it. Edges are added to the graph to model the assignments in lines 4 and 5. The effect of line 6 is to add another edge from the node labeled x. As the algorithm is flow-insensitive and conservative, edges can be added but not removed. Such changes are called *weak updates*. By analyzing the final graph, we can conclude that a and b point to different locations, but we can only conclude that x and y *may* point to the same location. □

Flow-insensitive analyses are efficient but imprecise. For programs with significant pointer manipulation, the imprecision snowballs and most edges may point to the same memory location. *Canonical abstractions* are a more precise abstract domain designed to address these issues [101]. The concrete domain is a graph-based representation of the heap augmented with logical predicates, which describe relationships between heap cells. Examples of such relationships are that one cell is reachable from the other, or that a cell does not point to the other. A canonical abstraction is an abstract graph with nodes representing heap cells, *summary nodes* representing multiple heap cells, and three-valued predicates that may evaluate to True, False or Don't Know. Thus, it is possible to distinguish between pointer variables that point to the same location, that do not point to the same location and that *may* point to the same location. More sophisticated relationships can be expressed and analyzed using predicates.

E. Static Analysis Tools

An early, popular static analysis tool for finding simple errors in C programs is LINT, released in 1979 by Bell Labs. Several modern tools emulate and extend LINT in terms of the kind of errors detected, warnings provided and user experience. FINDBUGS for Java is a notable modern tool with similar features. We mention LINT and FINDBUGS because of their significance and influence on static analysis tools. However, these tools are unsound and provide no rigorous guarantees so we do not discuss them further.

Though several commercial static analyzers are available, the details of the techniques implemented and their soundness is unclear, so we refrain from a full market survey, and only list a few exemplary tools. Grammatech Inc. produces CODESONAR, a tool using inter-procedural analyses to check for template errors in C/C++ code. These include buffer overflows, memory leaks, redundant loop and branch conditions. The K7 tool from KlocWork has similar features and supports Java.

The company Coverity produces PREVENT, a static analyzer, and EXTEND, a tool for enforcing coding standards. PREVENT has capabilities similar to those of CODESONAR, but also supports Microsoft COM and Win32 APIs and concurrency implemented using PThreads, Win32 or WindRiver VxWorks. In January 2006, as part of a United States Department of Homeland Security sponsored initiative, Coverity tools were used to find defects in over 30 open source projects.

Abstract interpretation tools were used to identify the error leading to the failure of the Ariane 5 rocket [88]. Abstract domains for finding buffer overflows and undefined results in numerical operations are implemented in the Astrée static analyzer [25], used to verify Airbus flight control software. Polyspace Technologies markets C and Ada static analyzers. The C Global Surveyor (CGS), developed at NASA, is a static analyzer specially developed for space mission software. The Polyspace tool and CGS have been used to analyze software used on the Mars Path-Finder, Deep Space One and Mars Exploration Rover. AbsInt GmbH is a company selling PAG, a program analyzer generator used to analyze architecture-dependent properties such as worst case execution time, cache performance, stack usage and pipeline behavior.

Several tools require annotations such as types, pre- and post-conditions, and loop invariants to be provided. Annotations may reduce the information the static analyzer has to compute and improve its precision, but increase the burden on the programmer. This approach is used to varying degrees in the (ESC/Java) tool family [61] and Microsoft's PREFIX and PREFAST [89]. The annotation and verification mechanism has been integrated in programming languages and development environments such as SPARK [16] and SPEC# [17], to make writing annotations a natural part of programming. We only mention these techniques for completeness.

F. Merits and Shortcomings

The experience of using static analyzers is comparable to using a compiler. Most analyzers can be used to analyze large software systems with minimal user interaction. Such tools are extremely robust, meaning that they can cope with large and varied inputs, and are efficient. Conversely, the properties that can be proved are often simple and are usually hard coded into the tools, for example, ensuring that array bounds are not exceeded, that arithmetic overflows do not occur, or, more recently, that assertion are not violated. Early static analyzers produced copious warnings and hence fell into disuse. Recent tools include options for the user to control the verbosity of the output and to specify which properties must be analyzed. Unlike in model checking, generating counterexamples is difficult or even impossible, due to the precision loss in join and widening operations, and is a current research topic [65].

Simple abstract domains and analyses, which disregard program structure may be useful for compiler optimization but rarely suffice for verification. The invariants computed by flow- and context-insensitive analyses can only be used to show the absence of simple errors. In contrast, model checking tools can prove more complicated properties of programs expressed in temporal or other logics, are more precise, and provide counterexamples. Model checkers are less robust due to the state-space explosion problem. These differences are discussed in greater detail in the next section.

III. SOFTWARE MODEL CHECKING

A. Introduction and overview

Model checking is an algorithmic method for determining if model of a system satisfies a correctness specification [36], [97]. A model of a program consists of *states* and *transitions*. A specification or *property* is a logical formula. A state is an evaluation of the program counter, the values of all program variables, and the configurations of the stack and the heap. Transitions describe how a program evolves from one state to another. Model checking algorithms exhaustively examine the reachable states of a program. This procedure is guaranteed to terminate if the state space is finite. If a state violating a correctness property is found, a *counterexample* – an execution trace demonstrating the error – is produced. Due to their diagnostic value, counterexamples, to quote Clarke and Veith, "... are the single most effective feature to convince system engineers about the value of formal verification" [35].

Model checking tools verify *partial* specifications, usually classified as *safety* or *liveness* properties. Intuitively, safety properties express the unreachability of bad states, such as those in which an assertion violation, null pointer dereference, or buffer overflow has occurred, or API usage contracts, like the order of function calls, are not respected. Liveness properties express that something good eventually happens, like the condition that requests must be served eventually, or that a program must eventually terminate.

The distinction between model checking and static analysis is primarily historical. Static analysis methods were used to compute simple, pre-defined facts about programs by analyzing the source code. These methods traded precision for efficiency, in particular by being flow- and path- insensitive, using abstraction, and by merging abstract states at join points. In contrast, model checking was conceived to check possibly complex, temporal logic properties of manually constructed, finite-state models. Model checkers emphasized precision, exploring a program's state space in a flow- and path-sensitive manner, without merging states. However, theoretical results have shown that static analysis methods can be cast as model checking algorithms and vice versa [102], [104]. In practice, static analyzers and model checkers still differ in their applicability and capabilities. Nonetheless, modern static analyzers support specification mechanisms, and software model checkers use abstraction and operate on program code, so the practical distinction may cease to be meaningful as well [23].

We discuss the two general approaches for state exploration in § III-B, and cover a popular technique for constructing abstract models in § III-C. We conclude this section with a survey of model checkers in § III-D and a discussion in § III-E.

B. Explicit and Symbolic Model Checking

The principal issue in model checking is *state-space explosion* [51]: the state-space of a software program is exponential in various parameters such as the number of variables and the width of datatypes. In the presence of function calls and dynamic memory allocation, it is even infinite. Concurrency exacerbates the problem because the different thread schedules, called *interleavings*, which must be considered are exponential in the number of statements. Model checking algorithms use instructions in the program to generate sets of states to be analyzed. These states must be stored to ensure that they are visited at most once. Methods for representing states are divided into two categories. *Explicit-state model checking* algorithms directly index states, and use graph algorithms to explore the state space, starting from the initial states. *Symbolic model checking* algorithms use implicit representations of sets of states and may start from the initial states, error states, or the property. We briefly describe both techniques.

Explicit state methods construct a state transition graph by recursively generating successors of initial states. The graph may be constructed in a depth-first, breadth-first, or heuristic manner. New states are checked for a property violation *on-the-fly*, so that errors can be detected without building the entire graph. Explored states are compressed and stored in a hash table to avoid recomputing their successors. If the available memory is insufficient, lossy compression methods can be used. *Bitstate hashing* or *hash compaction* uses a fixed number of bits from the compressed image of a state [74]. This may cause hash collisions, which lead to error states being missed. In practice, with state spaces containing close to a billion states, and hash tables of several hundred megabytes, the probability of missing a state can be less than 0.1%.

Partial order reduction is a method to prune the state space exploration of concurrent programs [62]. The order in which instructions in different threads are executed may not matter for proving some properties. Transitions whose interleavings do not affect the property can be grouped into classes. A model checker only needs to generate one representative of each class while constructing the state graph. In the best case, partial order reduction can reduce the state space to be explored by a factor that grows exponentially in the number of threads.

Symbolic model checking methods represent sets of states, rather than enumerating individual states. Common symbolic representations are BDDs [27] and propositional logic for finite sets [24], and finite automata for infinite sets [82]. The method enabled the verification of hardware designs with over 10^{20} states [90]. In contrast, explicit state methods at the time scaled to a few thousand states.

A BDD is obtained from a Boolean decision tree by maximally sharing nodes and eliminating redundant nodes. For a fixed variable ordering, BDDs are canonical, permitting Boolean function equivalence, essential in model checking, to be checked efficiently. However, BDDs grow very large. The issues in using finite automata for infinite sets are analogous. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time.

Symbolic techniques work well for proving correctness and handling state-space explosion due to program variables and data types. Explicit-state techniques are well suited to error detection and handling concurrency. An orthogonal approach to counter state-space explosion is *abstraction*. It suffices to analyze a sound abstraction of the program, with a smaller state space. Abstractions were manually constructed, but are constructed automatically in recent tools. Automated abstraction is based on abstract interpretation, but due to algorithmic differences, we discuss these methods separately in § III-C.

C. Predicate Abstraction

Promoted by the success of the SLAM toolkit [10], *predicate abstraction* is currently the predominant abstraction technique in software model checking. Graf and Saïdi use *logical predicates* to construct an abstract domain by partitioning a program's state space [64]. This process differs from standard abstract interpretation because the abstraction is parametrized by, and specific to, a program. The challenge in predicate abstraction is identifying predicates, since they determine the accuracy of the abstraction. In *counterexample-guided abstraction refinement* [87], [37], [7] (CEGAR), if model checking the abstraction yields a counterexample that does not exist in the concrete program, the abstract counterexample is

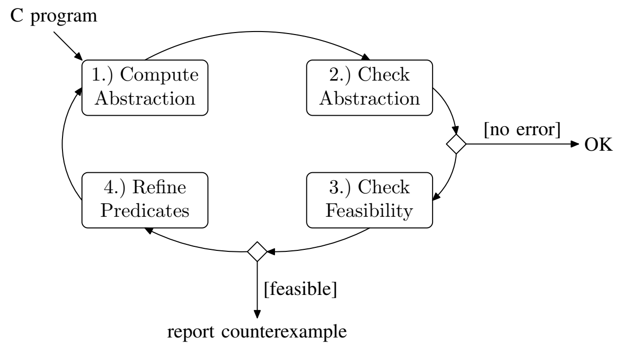


Fig. 2. The CEGAR Abstraction-Refinement Scheme as described in Section III-C

used to identify new predicates, and to obtain a more precise abstraction. Fig. 2 shows the four phases of the CEGAR loop: Abstraction, Verification, Simulation, Refinement. These phases are discussed in § III-C1, III-C2, III-C3, and III-C4, respectively. We use the program in Fig. 3(a), which is inspired by a typical buffer overflow bug, as a running example to illustrate these steps.

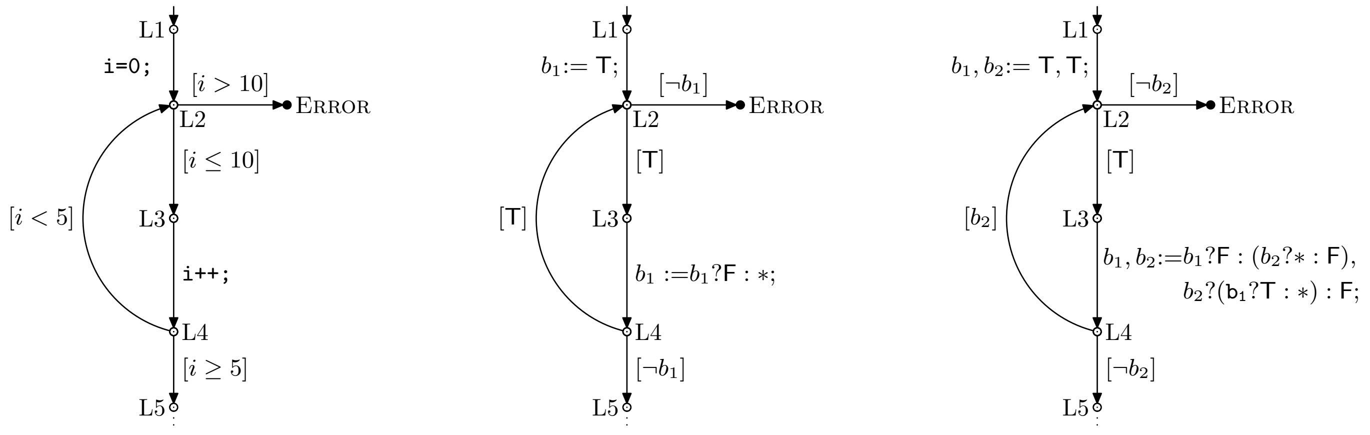
1) Abstraction techniques: In imperative programming languages, a program is a sequence of locations with instructions (e.g. $L1, L2, \ldots$ in Fig. 3(a)). The effect of an instruction $L1$ is modeled by a mathematical relation R_{L1}, which relates program states to their successors reached by executing $L1$. The union of the relations $R_{L1}, R_{L2}, \ldots$, say R, is the *transition relation* of the program, which is used for model checking.

In predicate abstraction, a sound approximation $\hat{R}$ of R is constructed using predicates over program variables. A predicate P partitions the states of a program into two classes: one in which P evaluates to true, and one in which it evaluates to false. Each class is an *abstract state*. Let A and B be abstract states. A transition is defined from A to B (i.e. $(A, B) \in \hat{R}$) *if there exists* a state in A with a transition to a state in B. This construction yields an *existential abstraction* of a program, sound for reachability properties [34]. The abstract program corresponding to $\hat{R}$ is represented by a *Boolean program* [14]; one with only Boolean data types, and the same control flow constructs as C programs (including procedures). Together, n predicates partition the state space into 2^n abstract states, one for each truth assignment to all predicates.

Example 4: The program from Example 1 with transition from $L3$ to $L4$ modified to `i++` is shown in Fig. 3(a). The abstraction is constructed using the predicate $(i = 0)$. The predicate's value at each program location is represented by the variable b_1 in Fig. 3(b). Executing `i++` in a state satisfying $(i = 0)$ will lead to a state satisfying $\neg(i = 0)$. In a state satisfying $\neg(i = 0)$, we do not know the value of the predicate after this instruction. Let $*$ denote that a value is either true or false. The effect of `i++` is abstractly captured by the conditional assignment to b_1 in Fig. 3(b).

Observe that every location reachable in the original program is reachable in the abstraction. Though the original program contains only one path to $L5$, the abstraction in Fig. 3(b) contains infinitely many such paths. □

Abstractions are automatically constructed using a decision



(a) Concrete Program (b) Abstraction for $P = \{b_1\}$ (c) Abstraction for $P = \{b_1, b_2\}$

Fig. 3. Iterative application of predicate abstraction: The first abstraction (b) of the original program (a) uses the predicate $(i = 0)$ (represented by the variable b_1). The second graph (c) shows a refinement of (b) with the additional predicate $(i < 5)$ (variable b_2).

procedure to decide, for all pairs of abstract states A, B, and instructions Li, if Li permits a transition from A to B.

Example 5: Consider the transition from $L3 \rightarrow L4$ in Fig. 3(c) abstracted using $(i = 0)$ and $(i < 5)$. A transition from the abstract state $\neg(i = 0) \wedge (i < 5)$ to $\neg(i = 0) \wedge \neg(i < 5)$ is permitted, and captured by the parallel assignment to b_1 and b_2. There is *no* transition from $\neg(i = 0) \wedge \neg(i < 5)$ to $\neg(i = 0) \wedge (i < 5)$ because this is inconsistent with `i++`.[1] □

As n predicates lead to 2^n abstract states, the method above requires $(2^n)^2$ calls to a decision procedure to compute an abstraction. In practice, a coarser but more efficiently computed *Cartesian Abstraction* is obtained by constructing an abstraction for each predicate separately and taking the product of the resulting abstract relations. The decision procedures are either First Order Logic theorem provers combined with theories such as machine arithmetic, for reasoning about the C programming language (e.g., ZAPATO [9] or Simplify [53]), or SAT-solvers, used to decide the satisfiability of a bit-level accurate representation of the formulas [39], [42], [85].

2) Verification: We now describe how an abstraction can be verified. Despite the presence of a potentially unbounded call stack, the reachability problem for sequential Boolean programs is decidable [29].[2] The intuition is that the successor of a state is determined entirely by the top of the stack and the values of global variables, both of which take values in a finite set. Thus, for each procedure, the possible pairs of input-output values, called *summary edges*, is finite and can be cached, and used during model checking [14], [60].

All existing model checkers for Boolean programs are symbolic. BDD-based tools do not scale if the number of variables is large. SAT-based methods scale significantly better, but cannot be used to detect fixed points. For this purpose, Quantified Boolean Formulas (QBF) solvers must be used. QBF, a classical PSPACE-complete problem, faces the same scalability issues as BDDs. The verification phase is therefore often the bottleneck of predicate abstraction.

[1]We ignore a potential overflow in favor of a simplified presentation.

[2]In fact, all ω-regular properties are decidable for sequential Boolean programs [26].

Example 6: We partially demonstrate the reachability computation with the abstract program in Fig. 3(c). After the transition from $L1 \rightarrow L2$, the variables b_1 and b_2 both have the value T. A symbolic representation of this abstract state is the formula $b_1 \wedge b_2$. After the transition $L3 \rightarrow L4$, the abstract state is $\neg b_1 \wedge b_2$. Thus, the possible states at the location $L2$ are now $(b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2)$, which equals b_2. After an additional iteration of the loop $L2$, $L3$, $L4$, the possible states are represented by T (meaning all values are possible). □

3) Simulation: The reachability computation above may discover that an error state is reachable in the abstract program. Subsequently, a *simulation step* is used to determine if the error exists in the concrete program or if it is *spurious*.

Example 7: Consider Fig. 3(b). The location ERROR is reachable via the execution trace $L1$, $L2$, $L3$, $L4$, $L2$, ERROR. This trace is spurious in the program in Fig. 3(a), because when $L2$ is reached for the second time, the value of i is 1 and the guard $i > 10$ prevents the transition to ERROR. □

Symbolic simulation, in which an abstract state is propagated through the sequence of program locations ocurring in the abstract counterexample, is used to determine if an abstract counterexample is spurious. If so, the abstraction must be *refined* to eliminate the spurious trace. This approach *does not* produce false error messages.

4) Refinement: There are two sources of imprecision in the abstract model. *Spurious traces* arise because the set of predicates is not rich enough to distinguish between certain concrete states. *Spurious transitions* arise because the Cartesian abstraction may contain transitions not in the existential abstraction. Spurious traces are eliminated by adding additional predicates, obtained by computing the weakest precondition (or strongest postcondition) of the instructions in the trace. Spurious transitions are eliminated by adding constraints to the abstract transition relation. The next example shows how new predicates are added.

Example 8: The instructions corresponding to the trace in Ex. 7 are `i=0`, $[i \leq 10]$, `i++`, $[i < 5]$, and $[i > 10]$ (see Fig. 3(a)). Computing the weakest precondition along this trace yields the formula $(i > 10) \wedge (i < 5)$. Using the strongest postcondition, we get $(i = 1) \wedge (i \geq 5)$. Both formulas are inconsistent proving that the trace is infeasible. Therefore, adding the predicates $(i = 1)$ and $(i < 5)$ to the abstraction is sufficient to eliminate this spurious trace. □

Existing refinement techniques heuristically identify a small set of predicates that explain the infeasibility of the counterexample. An alternative method is *Craig interpolation* [68].

Example 9: Adding the predicate $(i < 5)$, whose value is represented by b_2 yields the *refined* abstract program in Fig. 3(c). Reachability analysis determines that $\neg b_1 \wedge b_2$ and $b_1 \wedge b_2$ are the only reachable states at $L2$ in Fig. 3(c), hence ERROR is unreachable. The abstraction is sound, so we may conclude that the original program in Fig. 3(a) is safe. □

In fact, the predicate $(i < 5)$ in Ex. 9 suffices to show that any path ending with the suffix $L4$, $L2$, ERROR is infeasible, and consequently that ERROR is unreachable in Fig. 3(a). The predicate $(i = 0)$ is not necessary. Choosing refinement predicates "wisely" may lead to fewer refinement iterations.

Spurious transitions are eliminated by adding constraints to the abstract model. For instance, the transition in Fig. 3(c) from the abstract state $\neg b_1 \wedge b_2$ at $L3$ to $b_1 \wedge \neg b_2$ at $L4$ leads to the inconsistent state $(i = 0) \wedge (i \geq 5)$. Such transitions are eliminated by restricting the valuations of the Boolean variables before and after the transition (e.g., by adding the constraint $\neg(\neg b_1 \wedge b_2 \wedge b'_1 \wedge \neg b'_2)$, where the primed variables refer to the variables after executing the transition) [8].

Various techniques to speed up the refinement and the simulation steps have been proposed. *Path slicing* eliminates instructions from the counterexample that do not contribute to a property violation [78]. *Loop detection* is used to compute the effect of arbitrary iterations of loops in a counterexample in a single simulation step [86]. The refinement step can be accelerated by adding statically computed invariants [77], [21], including those that eliminate a whole class of spurious counterexamples [22]. Proof-based refinement eliminates all counterexamples up to a certain length, shifting the computational effort from the verification to the refinement phase, and decreasing the number of iterations required [3].

Concurrency: Multi-threaded programs pose a formidable challenge in software verification. Predicate abstraction alone is insufficient because the reachability problem for asynchronous Boolean programs is undecidable [99]. A model checker may (a) examine interleaved execution traces, or (b) use rely-guarantee reasoning [80]. In the former case, algorithms either compute an over-approximation of the set of reachable states [44], or restrict the number of context switches between threads [96]. Due to the undecidability issue, these methods are necessarily incomplete.

Rely-guarantee reasoning permits modular verification, by verifying each thread individually using environment assumptions that summarizes the behavior of all other threads. In our setting, predicate abstraction and counterexample-guided abstraction refinement is used to obtain an abstract model of the program threads and their environment assumptions [69].

D. Model Checking Tools

1) Model checking: Holzmann's SPIN project pioneered explicit-state software model checking [73], [75]. SPIN was initially used for verification of temporal logic properties of communication protocols specified in the PROMELA language. PROMELA supports simple data types, non-deterministic assignments and conditionals, simple loops, thread creation, and message passing. SPIN operates on-the-fly and extensively uses bit-state hashing and partial order reduction.

Some software model checkers, such as an early version of the Java Pathfinder (JPF), translate Java code to PROMELA and use SPIN for model checking [105]. PROMELA lacks important features like dynamic memory allocation and is therefore not well suited to modeling the Java language. Recent versions of JPF analyze the bytecode of Java programs directly and handle a much larger class of Java programs than the original implementation. JPF also supports symbolic techniques, but only for software testing purposes. The Bandera tool supports state abstraction, but does not fully automate it [57].

Besides SPIN and JPF, two prominent representatives of the class of explicit-state software model checkers are CMC [94] and Microsoft Research's ZING [5].

The VERISOFT software verification tool attempts to eschew state explosion by *discarding* the states it visits [63]. Since visited states are not stored, they may be repeatedly visited and explored. This method is *state-less* and has to limit the depth of its search to avoid non-termination. This approach is incomplete for transition systems that contain cycles.

2) Predicate Abstraction: The success of predicate abstraction for software model checking was initiated by the Microsoft Research's SLAM toolkit [10]. SLAM checks a set of about 30 predefined, system specific properties of Windows Device drivers, such as "a thread may not acquire a lock it has already acquired, or release a lock it does not hold". SLAM comprises the predicate abstraction tool C2BP [12], [11], the BDD-based model checker BEBOP [14] for Boolean programs [15], and the simulation and refinement tool NEWTON [13]. The BDD-based model checker MOPED [58] can be used in place of BEBOP to check temporal logic specifications.

An incarnation of SLAM, the Static Driver Verifier (SDV) tool, is currently part of a beta of the Windows Driver Development Kit (DDK).[3] When combined with Cogent [42], a decision procedure for machine arithmetic,SLAM can verify properties that depend on bit-vector arithmetic.

The tool BLAST uses *lazy abstraction*: The refinement step triggers the re-abstraction of only relevant parts of the original program [70]. The tighter integration of the verification and refinement phases enables a speedup of the CEGAR iterations. Unlike SLAM, BLAST uses Craig interpolation to derive refinement predicates from counterexamples [68]. Like SLAM, BLAST provides a language to specify reachability properties.

The verification tools mentioned above use general purpose theorem provers to compute abstractions and BDDs for model checking. SATABS uses a SAT-solver to construct abstractions and for symbolic simulation of counterexamples [39]. The bit-level accurate representation of C programs makes it possible to model arithmetic overflow, arrays and strings. SATABS automatically generates and checks proof conditions for array bound violations, invalid pointer dereferencing, division by

[3]http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

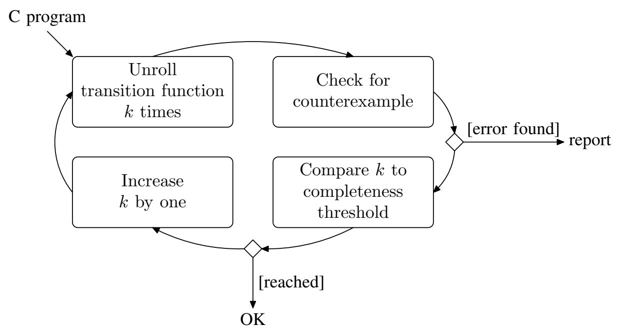


Fig. 4. High level overview of BMC

zero, and assertions provided by the user.

SATABS uses a SAT-based model checker BOPPO to compute the reachable states of the abstract program. BOPPO relies on a QBF-solver for fixed point detection [43]. SATABS can verify concurrent programs that communicate via shared memory. To deal with concurrency, BOPPO combines symbolic model checking with partial order reduction [43]. Unfortunately, this approach still has severe scalability issues [107].

Gurfinkel's model checker YASM can construct sound abstractions for liveness properties [66]. The SLAM-based TERMINATOR tool also checks liveness properties [45]. However, the latter is not dependent on predicate abstraction and can be based upon any software model checker.

Sagar Chaki's MAGIC framework [31] uses a compositional approach and decomposes the program into several smaller components which are verified separately. Furthermore, MAGIC is able to check *concurrent* programs that use message passing, but does not support shared memory.

In an experimental version of SLAM for concurrent programs, BEBOP can be replaced by either the explicit state model checker ZING [5] or the SAT-based tool BOPPO [44]. ZING does not report spurious counterexamples, but is potentially non-terminating. BOPPO handles asynchronous programs with recursion or an infinite number of threads by over-approximating the set of reachable states, at the cost of potential false positives.

E. Merits and Shortcomings

In practice, the counterexamples provided by model checkers are often of more value than a proof of correctness. Predicate abstraction in combination with CEGAR is suitable for checking control-flow related safety properties. Though no false positives are reported, the abstraction-refinement cycle may not terminate. Furthermore, the success of the approach depends crucially on the refinement step: Many existing refinement heuristics may yield a diverging set of predicates [79]. The main field of application is currently verification of safety properties of device drivers and systems code up to 50 kLOCs. Predicate abstraction does not work well in the presence of complex heap-based data structures or arrays.

IV. BOUNDED MODEL CHECKING

A. Background on BMC

Bounded model checking (BMC) is one of the most commonly applied formal verification techniques in the semiconductor industry. The technique owes this success to the impressive capacity of propositional SAT solvers. It was introduced in 1999 by Biere et al. as a complementary technique to BDD-based unbounded model checking [24]. It is called *bounded* because only states reachable within a bounded number of steps, say k, are explored. In BMC, the design under verification is unwound k times and conjoined with a property to form a propositional formula, which is passed to a SAT solver (Fig. 4). The formula is satisfiable if and only if there is a trace of length k that refutes the property. The technique is inconclusive if the formula is unsatisfiable, as there may be counterexamples longer than k steps. Nevertheless, the technique is successful, as many bugs have been identified that would otherwise have gone unnoticed.

Let R denote the transition relation of a design, containing current and next state pairs, I denote the initial state predicate, and p denote the property of interest. To obtain a BMC instance with k steps, the transition relation is replicated k times. The variables in each replica are renamed such that the next state of step i is used as current state of step $i+1$. The transition relations are conjoined, the current state of the first step is constrained by I, and one of the states must satisfy $\neg p$:

$$\begin{tikzcd}
I & \wedge R & \wedge & R & \wedge & & \wedge & R & \\
\bullet \arrow[rr] & & \bullet \arrow[rr] & & \bullet & \cdots & \bullet \arrow[rr] & & \bullet \\
\neg p & \vee & \neg p & \vee & \neg p & \vee & \neg p & \vee & \neg p
\end{tikzcd}$$

A satisfying assignment to this formula corresponds to a path from the initial state to a state violating p. The size of this formula is linear in the size of the design and in k.

We describe BMC for software in § IV-B and optimizations to deal with loops in § IV-C. BMC is not *complete*, that is, it cannot be used to prove and disprove properties, as we discuss in § IV-D. In § IV-E, we survey solvers for BMC formulas, followed by BMC tools in § IV-F, and conclude this section with a discussion in § IV-G.

B. Unwinding the Entire Program at Once

BMC is also applicable to system-level software. The most straight-forward manner to implement BMC for software is to treat the entire program as a transition relation as described in § III-C1. Each basic block is converted into a formula by transforming it into Static Single Assignment (SSA) form [2]. Arithmetic operators in basic blocks are converted into their circuit equivalents. Arrays and pointers are treated as in the case of memories in hardware verification using a large case split over the possible values of the address.

An unwinding with k steps permits exploring all program paths of length k or less. The size of this basic unwinding is k times the size of the program. For large programs, this is prohibitive, and thus, several optimizations have been proposed. The first step is to analyze the possible control flow in the program. Consider the small control flow graph in Fig. 5(a). Each node corresponds to one basic block; the edges correspond to possible control flow between the blocks. Note that block L1 can only be executed in the very first step of any path. Similarly, block L2 can only be executed in step 2, 4, 6 and so on. This is illustrated in Fig. 5(b): the unreachable nodes in each time-frame are in gray.

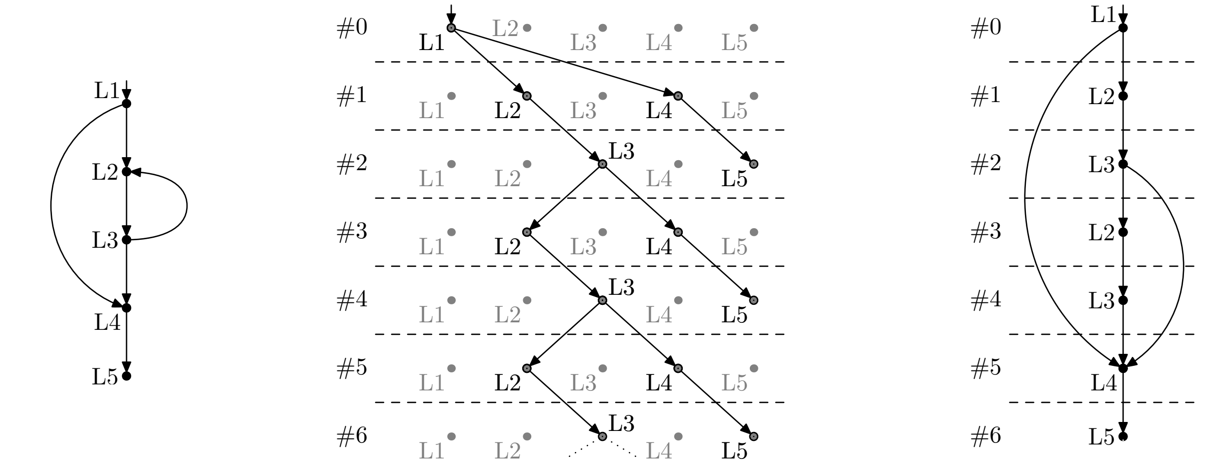


Fig. 5. There are two ways to unroll the model given by the control flow graph (a): When unrolling the entire transition relation (b), the unreachable nodes (in gray) may be omitted. When unrolling the loops (c), a separate unwinding bound has to be provided for each loop.

```
while(x)              if(x) {
  BODY;        ⟶        BODY;
                        if(x)
                          BODY;
                        else
                          assume(false);
                      }
```

Fig. 6. A loop-based unrolling of a **while** loop with depth two. The assume statement cuts of any path passing through it.

C. Unwinding Loops Separately

Consider again the example in Fig. 5(a): observe that any path through the CFG visits L4 and L5 at most once. Note that the unwinding of the transition relation in Fig. 5(b) contains three copies of L4 and L5. Such redundancy can be eliminated by building a formula that follows a specific path of execution rather than treating the program as a transition relation. In 2000, Currie et al. proposed this approach in a tool that unwinds assembly programs running on DSPs [49]

This motivated the idea of *loop unwinding*. Instead of unwinding the entire transition relation, each loop is unwound separately. Syntactically, this corresponds to a replication of the loop body together with an appropriate guard (Fig 6). The effect on the control flow graph is illustrated in Fig. 5(c), in which the loop between L2 and L3 is unwound twice. Such an unwinding may result in more compact formulas and requires fewer case-splits in the formula, as there are fewer successors for each time-frame. As a disadvantage, the loop-based unwinding may require more time-frames to reach certain locations. In Fig 5(b), the unwinding of depth 1 suffices to determine if $L4$ is reachable in one step, while an unwinding of depth 5 is required in Fig 5(c).

Loop unwinding differs from enumerative, path-based exploration: in the example, the path that corresponds to the branch from L1 to L4 merges with the path that follows the loop. As a consequence, the formula that is generated is linear in the depth and in the size of the program, even if there is an exponential number of paths through the CFG.

D. A Complete BMC for Software

Bounded Model Checking, when applied as described above, is inherently incomplete, as it searches for property violations only up to a given bound and never returns ”`No Errors`”. Bugs that are deeper than the given bound are missed. Nevertheless, BMC can be used to *prove* liveness and safety properties if applied in a slightly different way.

Intuitively, if we could search *deep enough*, we could guarantee that we have examined all the relevant behavior of the model, and that searching any deeper only exhibits states that we have explored already. A depth that provides such a guarantee is called a *completeness threshold* [84]. Computing the smallest such threshold is as hard as model checking, and thus, one settles for over-approximations in practice.

In the context of software, one way to obtain a depth-bound for a program is to determine a high-level worst-case execution time (WCET). This time is given by a bound on the maximum number of loop-iterations and is usually computed via a simple syntactic analysis of loop structures. If the syntactic analysis fails, an iterative algorithm can be applied. First, a guess of the bound on the number of loop iterations is made. The loop is then unrolled up to this bound, as in Fig 6, but with the assumption replaced by an assertion called an *unwinding assertion*. If the assertion is violated, there are paths in the program exceeding the bound, and a new guess for the bound is made [83], [38]. This method is applicable if the program (or its main loop body) has a run-time bound, which is highly desirable for many embedded applications.

In the next section, we discuss methods for reasoning about formulas arising in BMC.

E. Solving the Decision Problem

The result of unwinding, either of the entire program, by unwinding loops, or by following specific paths of execution, is a bit-vector formula. In addition to the usual arithmetic and bit-level operators, the formula may also contain operators related to pointers and (possibly unbounded) arrays. There is a large body of work on efficient solvers for such formulas. The earliest work on deciding bit-vector arithmetic is based on algorithms from the theorem proving community, and uses a canonizer and solver for the theory. The work by Cyrluk et al. [50] and by Barrett et al. on the Stanford Validity Checker [18] fall into this category. These approaches are very elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations (not inequalities) over bit-vectors.

With the advent of efficient propositional SAT solvers such as ZChaff [93], these approaches have been obsoleted. The most commonly applied approach to check satisfiability of these formulas is to replace the arithmetic operators by circuit equivalents to obtain a propositional formula, which is then passed to a propositional SAT solver. This approach is commonly called 'bit-blasting' or 'bit-flattening', as the word-level structure is lost. The Cogent [42] procedure mentioned earlier belongs to this category. The current version of CVC-Lite [20] pre-processes the input formula using a normalization step followed by equality rewrites before finally bit-blasting to SAT. Wedler et al. [106] have a similar approach wherein they normalize bit-vector formulas to simplify the generated SAT instance. STP [30], which is the engine behind EXE [109], is a successor to the CVC-Lite system; it performs several array optimizations, as well as arithmetic and Boolean simplifications on a bit-vector formula before bit-blasting. Yices [56] applies bit-blasting to all bit-vector operators except for equality. Bryant et al. present a method to solve hard instances that is based on iterative abstraction refinement [28].

F. Tools that implement BMC

There are a number of BMC implementations for software verification. The first implementation of a depth-bounded symbolic search in software is due to Currie et al. [49].

One of the first implementations of BMC for C programs is CBMC [83], [38], developed at CMU; it emulates a wide range of architectures as environment for the design under test. It supports both little- and big-Endian memory models, and the header files needed for Linux, Windows, and Mac-OS X. It implements loop unrolling as described in § IV-C, thus avoiding the exponential blowup inherent in path enumeration. It uses bit-flattening to decide the resulting bit-vector formula. It includes options to export the formula to various word-level formats. It is the only tool that also supports C++, SpecC and SystemC. The main application of CBMC is checking consistency of system-level circuit models given in C or SystemC with an implementation given in Verilog. IBM has developed a version of CBMC for concurrent programs [98].

The only tool that implements an unwinding of the entire transition system, as described in § IV-B, is F-SOFT [76], developed at NEC Research. It features a SAT solver customized for BMC decision problems. The benchmarks that have been reported are system-level UNIX applications such as `pppd`.

A number of variants of these implementations have been reported. Armando et al. implement a version of CBMC that generates a decision problem for an SMT solver for integer linear arithmetic [6]. The performance of off-the-shelf constraint solvers on such decision problems is evaluated in [40].

SATURN is a specialized implementation of BMC, tailored to the properties it checks [108]. It implements loop unwinding as described in § IV-C. The authors have applied it to check two different properties of Linux kernel code: NULL-pointer dereferences and locking API conventions. They demonstrate that the technique is scalable enough to analyze the entire Linux kernel. Soundness is relinquished for performance; SATURN performs at most two unwindings of each loop. Bugs that require more than two unwindings are missed.

The EXE tool is also specialized to bug-hunting [109]. It combines explicit execution and path-wise symbolic simulation to detect bugs in system-level software such as file system code. It uses a very low-level memory model, which permits checking programs that contain arbitrary pointer type-casts.

G. Merits and Shortcomings

BMC is the best technique to find shallow bugs, and it provides a full counterexample trace in case a bug is found. It supports the widest range of program constructions. This includes dynamically allocated data structures; for this, BMC does not require built-in knowledge about the data structures the program maintains. On the other hand, completeness is only obtainable on very 'shallow' programs, i.e., programs without deep loops.

V. SUMMARY AND CONCLUSION

In this article, we surveyed three main techniques for automatic formal verification of software. We focused on tools that provide some form of formal guarantee, and thus, aid to improve software quality. We summarize the tools discussed in this survey in Tab. I. The table provides the main features of each tool, including the programming languages it supports.

Static analysis techniques based on abstract interpretation scale well at the cost of limited precision, which manifests itself in a possibly large number of false warnings. The tool support is mature. Model Checking tools that use abstraction can check complex safety properties, and are able to generate counterexamples. Bounded Model Checkers are very strong at detecting shallow bugs, but are unable to prove even simple properties if the program contains deep loops. The model checking-based tools for software are less robust than static analysis tools and the market for these tools is in its infancy.

Research Challenges: The challenges for future research in software analysis are dynamically allocated data structures, shared-variable concurrency, and the environment problem. None of the tools we survey is able to assert even trivial properties of dynamically allocated data structures, despite the large body of (theoretical) research in this area. Similarly, concurrency has been the subject of research for decades, but the few tools that analyze programs with shared-variable concurrency still scale poorly. Any formal analysis requires a formal model of the design under test – in case of software, this model often comprises a non-trivial environment that the code runs in (libraries, other programs, and hardware components). As programs often rely on properties of this environment, substantial manual effort is required to model these parts of the environment.

	Tool name	Tool developer	Symbolic analysis	Abstraction	Counterexample	BMC	Concurrency	Languages
II	ASTRÉE	École Normale Supérieure	×	×				C (subset)
	CODESONAR	Grammatech Inc.	×	×				C, C++, ADA
	PolySpace	PolySpace Technologies	×	×			×	C, C++, ADA, UML
	PREVENT	Coverity	×	×			×	C, C++, Java
III	BLAST	UC Berkeley/EPF Lausanne	×	×	×		×	C
	F-SOFT (abs)	NEC	×	×	×			C
	Java PathFind.	NASA	×		×	×	×	Java
	MAGIC	Carnegie Mellon University	×	×	×		$\times^1$	C
	SATABS	Oxford University	×	×	×		×	C, C++, SpecC, SystemC
	SLAM	Microsoft	×	×	×		×	C
	SPIN	Bell Labs^2			×	×	×	PROMELA, C^3
	ZING	Microsoft Research			×	×	×	ZING (object oriented)
IV	CBMC	CMU/Oxford University	×		×	×		C, C++, SpecC, SystemC
	F-SOFT (bmc)	NEC	×		×	×		C
	EXE	Stanford University	×		×	×		C
	SATURN	Stanford University	×		×	×		C

1) does not support shared memory concurrency
2) originally developed by Bell Labs, now freely available
3) C is supported by automatic translation to PROMELA

TABLE I
TOOL OVERVIEW

ACKNOWLEDGEMENTS. We thank Michele Mazzucchi, Viktor Schuppan, and Thomas Wahl for their helpful input.

REFERENCES

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages (POPL)*, pages 1–11. ACM, 1988.

[3] N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 260–274. Springer, 2004.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[5] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Concurrency Theory (CONCUR)*, pages 1–15. Springer, August 2004.

[6] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking and Software Verification (SPIN)*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.

[7] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification (CAV)*, volume 697 of *LNCS*. Springer, 1993.

[8] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*. Springer, 2004.

[9] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*. Springer, 2004.

[10] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*. Springer, 2004.

[11] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213. ACM, 2001.

[12] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Software Tools for Technology Transfer (STTT)*, 5(1):49–58, 2003.

[13] T. Ball and S. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[14] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking and Software Verification (SPIN)*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[15] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[16] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[17] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[18] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM, June 1998.

[19] R. E. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[20] S. Berezin, V. Ganesh, and D. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, 2005.

[21] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.

[22] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Programming Language Design and Implementation (PLDI)*, pages 300–309. ACM, 2007.

[23] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software

verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 504–518. Springer, 2007.

[24] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[25] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 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*, volume 2566 of *LNCS*, pages 85–108. Springer, October 2002.

[26] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

[27] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[28] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*. Springer, 2007.

[29] J. R. Büchi. Regular canonical systems. *Archive for Mathematical Logic*, 6(3-4):91, April 1964.

[30] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

[31] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, pages 388–402, June 2004.

[32] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[33] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Symposium on Static Analysis (SAS)*, pages 312–327, 2004.

[34] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[35] E. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 208–224. Springer, 2003.

[36] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[37] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[38] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[39] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.

[40] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[41] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[42] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 296–300. Springer, 2005.

[43] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In *Model Checking and Software Verification (SPIN)*, volume 3639 of *LNCS*, pages 75–90. Springer, 2005.

[44] B. Cook, D. Kroening, and N. Sharygina. Over-approximating Boolean programs with unbounded thread creation. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 53–59. IEEE, 2006.

[45] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction-refinement for termination. In *Static Analysis Symposium (SAS)*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

[46] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[47] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282. ACM, 1979.

[48] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978.

[49] D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135. ACM, 2000.

[50] D. Cyrluk, M. O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification (CAV)*, LNCS, pages 60–71. Springer, 1997.

[51] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Compututer and System Sciences*, 72(4):547–575, 2006.

[52] N. Dershowitz. Software horror stories. In *http://www.cs.tau.ac.il/~nachumd/verify/horror.html*.

[53] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, January 2003.

[54] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Programming Language Design and Implementation (PLDI)*, pages 230–241. ACM, 1994.

[55] N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Symposium on Static Analysis (SAS)*, pages 194–212. Springer, 2001.

[56] B. Dutertre and L. de Moura. The Yices SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, September 2006.

[57] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering (ICSE)*, pages 177–187. IEEE, 2001.

[58] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 324–336. Springer, 2001.

[59] J. Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP)*, volume 2986 of *LNCS*. Springer, 2004.

[60] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Verification of Infinite State Systems (INFINITY)*, volume 9 of *ENTCS*. Elsevier, 1997.

[61] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, 2002.

[62] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[63] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages (POPL)*, pages 174–186. ACM, 1997.

[64] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[65] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 474–488. Springer, 2006.

[66] A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 212–226. Springer, 2006.

[67] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design (FMSD)*, 11(2):157–185, 1997.

[68] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.

[69] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.

[70] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[71] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[72] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[73] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[74] G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Model Checking and Software Verification (SPIN)*, 1997.

[75] G. J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.

[76] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *International Conference on Computer Design (ICCD)*, pages 297–308. IEEE, 2005.

[77] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 137–151. Springer, 2006.

[78] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47. ACM, 2005.

[79] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[80] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[81] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Principles of Programming Languages (POPL)*, pages 244–256, New York, NY, USA, 1979. ACM Press.

[82] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich ssertional languages. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 424–435. Springer, 1997.

[83] D. Kroening, E. M. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.

[84] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[85] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[86] D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 152–165. Springer, 2006.

[87] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[88] P. Lacan, J. N. Monfort, L. V. Q. Ribal, A. Deutsch, and G. Gonthier. ARIANE 5 – The Software Reliability Verification Process. In *ESA SP-422: DASIA 98 - Data Systems in Aerospace*, pages 201–205, 1998.

[89] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[90] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[91] M. Menasche and B. Berthomieu. Time petri nets for analyzing and verifying time dependent communication protocols. In *Protocol Specification, Testing, and Verification*, pages 161–172, 1983.

[92] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[93] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[94] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.

[95] P. Naur. Checking of operand types in ALGOL compilers. In *NordSAM 64; BIT 5*, pages 151–163, 1965.

[96] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[97] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, pages 337–351, 1982.

[98] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.

[99] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[100] J. C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

[101] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[102] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Principles of Programming Languages (POPL)*, pages 38–48, New York, NY, USA, 1998. ACM.

[103] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41. ACM, 1996.

[104] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software*, pages 346–365, London, UK, 1991. Springer.

[105] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering (ASE)*, 10(2):203–232, 2003.

[106] M. Wedler, D. Stoffel, and W. Kunz. Normalization at the arithmetic bit level. In *Design Automation Conference (DAC)*, pages 457–462. ACM, 2005.

[107] T. Witkowski, N. Blanc, G. Weissenbacher, and D. Kroening. Model checking concurrent Linux device drivers. In *Automated Software Engineering (ASE)*, pages 501–504. IEEE, 2007.

[108] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Principles of Programming Languages (POPL)*, pages 351–363. ACM, 2005.

[109] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy (S&P)*, pages 243–257. IEEE, 2006.

[110] S. Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, volume 1494 of *LNCS*, pages 114–152. Springer, 1996.

Vijay D'Silva received a Bachelors degree in Computer Science and Engineering from the Indian Institute of Technology Bombay, a Masters degree in Business and Computer Science from the University of Zurich, and is currently a PhD student at ETH Zürich, Switzerland.

Daniel Kroening received the M.E. and doctoral degrees in computer science from the University of Saarland, Saarbrücken, Germany, in 1999 and 2001, respectively. He joined the Model Checking group in the Computer Science Department at Carnegie Mellon University, Pittsburgh PA, USA, in 2001 as a Post-Doc.

He was an assistant professor at the Swiss Technical Institute (ETH) in Zürich, Switzerland, from 2004 to 2007. He is now a lecturer at the Computing Laboratory at Oxford University. His research interests include automated formal verification of hardware and software systems, decision procedures, embedded systems, and hardware/software co-design.

Georg Weissenbacher received a diploma in Telematics from Graz University of Technology, Austria, and is currently a doctoral student at ETH Zürich in Switzerland. His research is funded by Microsoft Research through its European PhD Scholarship programme.

Prior to his doctoral studies, he worked as a software developer at ARC Seibersdorf research and Joanneum Research, the largest non-academic research institutions in Austria.