Book Title
Book Editors
IOS Press, 2003

Grand Challenge: Model Check Software

Edmund Clarke, Himanshu Jain, Nishant Sinha
5000 Forbes Ave, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. Model checking has been successfully employed for verification of industrial hardware systems. Recently, model checking techniques have also enjoyed limited success in verifying software systems, viz., device drivers. However, there are several hurdles which must be overcome before model checking can be used to handle industrial-scale software systems. This article reviews some of the prominent model checking techniques being used for verification of software and summarizes the existing challenges in the field.

Keywords. Software model checking. Counterexample-guided abstraction refinement.

1. Introduction

Critical infrastructures in several domains, such as medicine, power, telecommunications, transportation and finance are highly dependent on computers. Disruption or malfunction of services due to software failures (accidental or malicious) can have catastrophic effects, including loss of human life, disruption of commercial activities, and huge financial losses. The increased reliance of critical services on software infrastructure and the dire consequences of software failures have highlighted the importance of software reliability, and motivated systematic approaches for asserting software correctness. While testing is very successful for finding simple, relatively shallow errors, it cannot guarantee that a program conforms to its specification. Consequently, testing by itself is inadequate for critical applications, and needs to be complemented by automated verification.

Model checking [15,19] is an automated technique to check the correctness of finite state concurrent systems. The verification procedure involves an exhaustive search of the state space of the system. As compared to theorem proving, model checking is faster primarily due to the fact that it does not involve generating tedious proofs, which often need manual intervention. Temporal logics are used to express a wide variety of partial system specifications, which are then verified by an appropriate model checking procedure. Finally, the technique is capable of providing diagnostic counterexamples which are useful for debugging the system during its design. Model checking has been successfully employed to verify and detect bugs in non-trivial hardware systems, e.g., IEEE Futurebus [20] etc.

Real-life software systems often involve a large number of processes and complex data structures. Model checking, when applied to such systems, encounters the infamous problem of state space explosion. Twenty five years of research have led to development of multiple techniques to counter this problem. Such techniques include using symbolic data structures, abstraction, partial-order reduction, compositional reasoning, symmetry reduction, slicing and semantic minimization. In particular, symbolic representation of state space using compact data structures like BDDs [9] and model abstraction are two of the techniques known to handle the state space explosion in hardware verification efficiently. Inspite of their widespread success in verifying hardware systems, symbolic representation alone is not sufficient to alleviate the problem in the case of software verification. For example, as opposed to hardware, the state space of software programs is potentially infinite due to presence of unbounded data structures and pointers. Abstraction techniques are therefore considered to be of increased importance for verifying software.

Abstraction techniques reduce the state space during verification essentially by ignoring details of the system description (variables, predicates on variables etc.) which are irrelevant for showing the correctness of a given property on the system. The framework of *existential* abstraction involves computing a small overapproximation of the original system such that validity of universal specifications on the abstract system implies their validity on the concrete model also. However, existential abstractions allow for false negatives (also known as spurious counterexamples) and require *refinement* of the previous coarse abstraction in order to eliminate them. Owing to the fact that abstractions are difficult to compute manually together with the possibility of false negatives, several *automated* abstraction refinement techniques have been proposed [38,18]. Clarke et al. developed an automated abstraction-refinement technique [18] which uses the spurious counterexamples obtained when model checking an abstraction, for the purpose of refining it. Counterexample-guided abstraction refinement (CEGAR) together with predicate abstraction [28,21,42] have been found to be effective in handling the large state spaces of software systems [4] and also forms the core idea behind several software model checking tools [43,32,12,35].

Although software verification has been the subject of ambitious projects for several decades, software verification tools have, until recently, not attained the level of practical applicability required by industry. Motivated by urgent industrial need, the success and maturity of formal methods in hardware verification, and by the arrival of new techniques such as predicate abstraction, several research groups have started to develop a new generation of software verification tools. A common feature of all these tools is that they operate directly on programs written in a general purpose programming language such as C or Java instead of those written in a more restricted modeling language such as Promela. In addition, all of them are characterized by a CEGAR-based model checking algorithm which interacts with theorem provers and decision procedures to reason about software abstractions, in particular about abstractions of data types.

In this paper, we first present the background about model checking and automated abstraction procedures, including the details of the CEGAR approach and theorems on property-preserving abstractions. We then briefly survey the

existing methods for software verification. Finally, we discuss a promising SAT-based [41] CEGAR scheme recently proposed by members of model checking group at CMU and its applications to software model checking.

2. Background

Kripke Structures. In model checking, the system to be verified is formally represented by a finite Kripke structure. Essentially, a Kripke structure is a directed graph whose vertices are labeled by sets of atomic propositions. Vertices and edges are called states and transitions respectively. One or more states are considered to be initial states. More formally, a Kripke structure over a set of atomic propositions A is a tuple $K = (S, R, L, I)$ where S is the set of states, $R \subseteq S^2$ is the set of transitions, $I \subseteq S$ is the non-empty set of initial states, and $L : S \rightarrow 2^A$ labels each state by a set of atomic propositions. Note that Kripke structures may be transformed into automata with labels on edges and vice-versa. We, therefore, use Kripke structures to represent both concrete and abstract state transition graphs of the system being verified.

Computation Tree Logic. Computation Tree Logic(CTL) is an extension of propositional logic obtained by adding path quantifiers and temporal operators.

1. **Path quantifiers:**
   A for every path
   E there exists a path

2. **Temporal Operators:**
   Xp p holds next time
   Fp p holds sometime in the future
   Gp p holds globally in the future
   p**U**q p holds until q holds

In the CTL each temporal operator must be immediately preceded by a path quantifier. Thus, CTL can be viewed as a temporal logic based on the compound operators **AX, EX, AF, EF, AG, EG, AU, EU**. Let s_0 be a state in a kripke structure K. The formal semantics of **EX**, **EG** and **EU** is defined as follows:

$s_0, K \models \mathbf{EX}\phi$ iff there exists a path $p = s_0, s_1, \ldots$ such that $s_1, K \models \phi$

$s_0, K \models \mathbf{EG}\phi$ iff there exists a path $p = s_0, s_1, \ldots$ such that
for all $i \geq 0$, $s_i, K \models \phi$

$s_0, K \models \mathbf{E}\phi\mathbf{U}\psi$ iff there exists a path $p = s_0, s_1, \ldots$ and an $i \geq 0$ such that
for all $0 \leq j \leq i$, $s_j, K \models \phi$ and $s_i, K \models \psi$

The remaining CTL operators are defined by abbreviations as follows: $\mathbf{EF}\phi = \mathbf{E}(true\mathbf{U}\phi), \mathbf{AG}\phi = \neg\mathbf{EF}\neg\phi$ and so on.

ACTL is the fragment of CTL where only the operators involving **A** are used, and negation is restricted to atomic formulas. An important feature of ACTL is the existence of counterexamples. For example, the CTL specification AFp denotes *on all paths, p holds sometime in the future.* If the specification AFp is violated, then there exists an infinite path where p never holds. This path is called a counterexample of AFp. In this paper, we will focus on counterexamples which

are finite or infinite paths. For a formal definition of CTL and related temporal logics such as LTL, please refer to [19].

Model Checking Problem. Given a Kripke structure $K = (S, R, I, L)$ and a specification ϕ in a temporal logic such as CTL, the model checking problem is the problem of finding all states s such that $s, K \vDash \phi$ and checking if the initial states are among these. An explicit state model checker is a program which performs model checking directly on a Kripke structure obtained from the program description.

State space explosion. Kripke structure represents the state space of the system under investigation which potentially is of size exponential in the size of the system description. For example, the number of reachable states of a concurrent program can grow exponentially with the number of components due to the large number of possible execution interleavings. Therefore, even for systems of relatively modest size, it is often impossible to compute their Kripke structures explicitly. The state space explosion problem in model checking remains the chief obstacle to the practical verification of real-world distributed systems.

Several techniques have been proposed to alleviate the state explosion problem [19]. In particular, symbolic techniques represent the state space compactly using data structures like Ordered Binary Decision Diagrams(OBDDs) [9]. Abstraction techniques, in contrast, use the model description and the property specification in order to abstract away the irrelevant state space. Abstractions are a promising technique to mitigate the state space explosion that occurs during verification of software systems.

2.1. Model Checking using Abstractions

Abstraction techniques reduce the program state space by mapping the set of states of the actual system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system. Abstractions are most often performed in an informal, manual manner, and require considerable expertise. The framework of existential abstractions provides a systematic way of computing *conservative* over-approximations of the concrete system.

Existential Abstraction. Intuitively speaking, existential abstraction amounts to partitioning the states of a Kripke structure into clusters, and treating the clusters as new abstract states. Formally, an abstraction function h is described by a surjection $h : S \rightarrow \hat{S}$ where $\hat{S}$ is the set of abstract states. The surjection h induces an equivalence relation $\equiv$ on the domain S in the following manner: let s, s' be states in S, then $s \equiv s'$ iff $h(s) = h(s')$. Since an abstraction can be represented either by a surjection h or by an equivalence relation $\equiv$, we sometimes switch between these representations.

The abstract Kripke structure $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$ corresponding to the abstraction function h is defined as follows:

1. $\hat{I}(\hat{s})$ iff $\exists s.\ h(s) = \hat{s} \wedge I(s)$.
2. $\hat{R}(\hat{s_1}, \hat{s_2})$ iff $\exists s_1, s_2.\ h(s_1) = \hat{s_1} \wedge h(s_2) = \hat{s_2} \wedge R(s_1, s_2)$.
3. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$.

An atomic formula f respects an abstraction function h if for all s and s' in the domain S, $(s \equiv s') \Rightarrow (s \vDash f \Leftrightarrow s' \vDash f)$. Let $\hat{s}$ be an abstract state. $\hat{L}(\hat{s})$

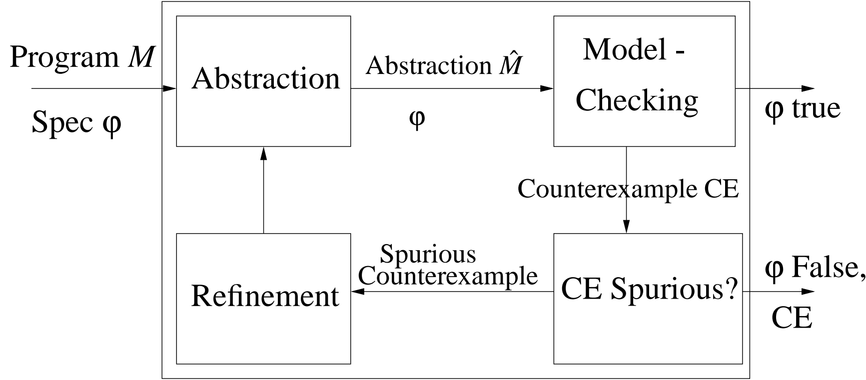

Figure 1. The Counterexample-based Abstraction Refinement(CEGAR) Framework

is *consistent*, if all concrete states corresponding to $\hat{s}$ satisfy all labels in $\hat{L}(\hat{s})$, i.e., collapsing a set of concrete states into an abstract state does not lead to contradictory labels.

Spurious Counterexamples. It is easy to see that $\hat{M}$ is a conservative approximation of M. Thus, model checking $\hat{M}$ may potentially lead to wrong results. The following theorem shows that at least for ACTL, the specifications which hold on $\hat{M}$, hold on M as well.

Theorem 1 *Let h be an abstraction function and ϕ be an ACTL specification where the atomic subformulas respect h. Then the following holds: (i) $\hat{L}(\hat{s})$ is consistent for all abstract states s in M; (ii) $\hat{M} \models \phi \Rightarrow M \models \phi$.*

2.2. Counterexample-Guided Abstraction

Recall that for a Kripke structure M and an ACTL formula ϕ, our goal is to check whether the Kripke structure $\hat{M}$ satisfies ϕ. Our methodology consists of the following main steps, cf. Figure 1.

1. Generate the initial abstraction: We generate an initial abstraction h by examining the transition blocks corresponding to the variables of the program which describes M. A detailed description of the initial abstraction approach is given in [18].
2. Model-check the abstract structure: Let $\hat{M}$ be the abstract Kripke structure corresponding to the abstraction function h. We check whether $\hat{M} \models \phi$. If the check is affirmative, then we can conclude that $M \models \phi$ (see Theorem 1). Suppose the check reveals that there is a counterexample CE. We then verify if CE is a valid counterexample,
3. Validate the counterexample: A counterexample is said to be valid (or actual) if it is an actual behavior of the concrete model M. If CE turns out to be an actual counterexample, we report it to the user. Otherwise CE is a spurious counterexample and we proceed to step 4.
4. Refine the abstraction: We refine the abstraction function h by partitioning one or more equivalence classes of $\equiv$ so that, after the refinement, the

abstract structure $\hat{M}$ corresponding to the refined abstraction function no longer admits the spurious counterexample CE. We will discuss SAT-based algorithms to achieve this in the later sections. After refining the abstraction function, we return to step 2.

Using counterexamples to refine abstract models has been investigated by a number of researchers beginning with the localization reduction approach of Kurshan [38]. Counterexample-based abstraction refinement (CEGAR) [18] forms the basis of several software verification tools, including SLAM [4], BLAST [32] and MAGIC [12]. Considerable progress in the domain of software verification over the last few years has been driven by the emergence of powerful yet automated abstraction techniques such as predicate abstraction [28]. Here, the initial abstractions are computed using a given set of seed predicates on program variables and the spurious counterexamples are removed by addition of new predicates, automatically inferred during the refinement phase. Before presenting the details of the CEGAR technique using predicate abstraction for software, we first briefly survey the existing software model checking techniques and tools.

3. Software Model Checking

Software model checking as compared to hardware model checking faces several new and difficult problems. These problems arise mainly due to the following constructs, common to several programming languages:

- Presence of large/unbounded base types, e.g., int, float.
- User-defined types and classes.
- Pointers/aliasing together with unbounded heap size.
- Procedure calls, recursion, function calls through pointers, dynamic method lookup.
- Concurrency together with unbounded number of threads
- Exceptions and callbacks.

The problems are further accentuated by the large code size of industrial software and unavailability of source code for system libraries and routines. Software model checking is a multifaceted and difficult task, and it is only to be expected that numerous approaches with different strengths and limitations will be required if the wide variety of existing industrial-scale programs are to be handled. We now present an overview of a set of useful software verification methodologies.

3.1. Survey of Current Approaches

Combining Static Analysis with Model Checking. Static analysis is used to extract a finite state model from the Boolean abstraction of a program obtained by predicate abstraction. Model checking is then used to verify a partial specification ϕ on this model. This technique has been used widely and forms the core of several model checkers for the C programming language: SLAM at Microsoft Research [4,43], Bandera [24,5] at KSU, JPF [8,37] at NASA, BLAST [32,7] at Berkeley and MAGIC [12,39] at CMU.

Symbolic Execution. This technique tries to simulate all the possible execution paths of a program using symbolic representation of the program variables, e.g., path predicates. Since the number of possible executions may be unbounded, the algorithm uses backtracking to prune away portions of state space which it considers irrelevant. Notable examples of tools using symbolic execution together with backtracking are Verisoft [27] and Prefix [10].

Bounded Model Checking. Kroening et al. have successfully used bounded model checking (BMC) [6] along with satisfiability checking (SAT) [41] to detect bugs in programs [16]. The key idea is to avoid the expensive fixpoint computation over the reachable state space by only considering the states reachable by exploring a fixed number of transitions, say n, from the initial state. The program transition relation is first unrolled up to the fixed depth symbolically and the resulting Boolean formula together with the error condition is passed to a SAT solver. If a satisfying assignment to the formula is found, a bug is detected. Otherwise, the depth n is increased and the algorithm proceeds with the next iteration.

Design using Statecharts and Esterel programming languages. The code is synthesized from finite state behavioral models of software, which avoid the complexity of actual software implementation while remaining sufficiently expressive.

Other techniques include use of finite state machines to look for patterns in the program control flow graph [26]. Some prominent software verification tools include:

- **SLAM:** Microsoft Research's SLAM project [4] is focused on verifying safety properties of sequential programs. They compute Boolean abstractions of programs using predicate abstraction and then perform the model checking of the Boolean programs followed by refinement (if needed). Procedures are handled by computing their summaries beforehand. The project has been very successful at analyzing device drivers. An ongoing project, ZING, aims to handle concurrent programs.
- **BLAST:** BLAST [32] is an explicit-state abstraction-based model checker originally developed for sequential programs. During the refinement stage, the tool adds predicates lazily, i.e., refines only relevant portions of the state space. The BLAST approach has also been extended to verify some concurrent programs [31].
- **MAGIC:** MAGIC [12], developed by members of the model checking group at CMU, is also an explicit-state abstraction-based model checker. MAGIC can be used to verify either simulation relation or trace containment. MAGIC's two-level abstraction [11] can be used to verify properties of concurrent message-passing systems in the CEGAR framework. Since the predicates are stored explicitly, both MAGIC and BLAST are able to avoid spurious counterexamples due to infeasible predicate valuations, which may occur in SLAM.
- **SPIN, JPF, and Bogor:** SPIN [33,44] is widely used to check properties of models of concurrent systems. Although SPIN was initially developed to verify distributed concurrent protocols, it now includes a frontend for translating C programs [34] into its input language, PROMELA. JPF [8] model checks Java source code, and handles most of the concurrency fea-

tures of Java. It is based on depth-first search of the state space and has an built in scheduler for handling Java threads. Bogor [25] is an extensible model checker for object-oriented systems, featuring a number of state space reductions. All these systems use explicit-state enumeration.

- **CBMC:** CBMC [16], also developed by members of the model checking group at CMU, is a tool based on Bounded Model Checking (BMC) [6]. CBMC supports the full ANSI-C standard for C programs, which includes keeping track of pointer arithmetic and overflows arising out of arithmetic operations.
- **F-Soft:** F-Soft [35] is an ongoing software verification project at NEC Laboratories. Verification is done using bounded model checking or iterative predicate abstraction and refinement. It uses a Boolean analysis system called DiVeR [29], which includes various SAT-based and BDD-based methods for performing both bounded and unbounded verification including BMC-based techniques for providing correctness proofs.

4. CEGAR using SAT-based Predicate Abstraction

Recall that in Counterexample Guided Abstraction Refinement (CEGAR) paradigm one starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm and depend on the abstraction and refinement techniques used. We discuss below a CEGAR loop where abstraction is done with respect to a finite set of predicates over the program variables.

Predicate abstraction [28,21] is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates. The CEGAR steps in the context of predicate abstraction are described below.

1. **Program Abstraction.** Given a set of predicates, a finite state model is extracted from the code of a software system and the abstract transition system is constructed.
2. **Verification.** A model checking algorithm is run in order to check if the model created by applying predicate abstraction satisfies the desired behavioral claim φ. If the claim holds, the model checker reports success (φ is *true*) and the CEGAR loop terminates. Otherwise, the model checker extracts a counterexample and the computation proceeds to the next step.
3. **Counterexample Validation.** The counterexample is examined to determine whether it is spurious. This is done by simulating the (concrete) program using the abstract counterexample as a guide, to find out if the counterexample represents an actual program behavior. If this is the case, the bug

is reported (φ is *false*) and the CEGAR loop terminates. Otherwise, the CEGAR loop proceeds to the next step.

4. **Predicate Refinement.** The set of predicates is changed in order to eliminate the detected spurious counterexample, and possibly other spurious behaviors introduced by predicate abstraction. Given the updated set of predicates, the CEGAR loop proceeds to Step 1.

We discuss the above steps in detail now. For illustration we will use a small language consisting of assignment and guarded goto commands. Let v denote a variable, *exp* denote an expression of same type as v, g denotes a guard (condition), and l denote a label. Let `v:=exp` denote an assignment statement and `if` g *then* `goto` l denote a guarded goto statement. The language we use for illustration is given as follows:

$$\texttt{v:=exp} \mid \texttt{if } g \texttt{ then goto } l$$

4.1. Predicate Abstraction

In predicate abstraction [28], the variables of the concrete program are replaced by Boolean variables that correspond to predicates on the variables in the concrete program. These predicates are functions that map a concrete state $\bar{v} \in S$ into a Boolean value. Let $\mathtt{B} = \{\pi_\mathtt{1}, \ldots, \pi_\mathtt{k}\}$ be the set of predicates over the given program. When applying all predicates to a specific concrete state $\bar{v}$, one obtains a vector of Boolean values, which represents an abstract state $\bar{b}$. We denote this function by $\alpha(\bar{v})$. It maps a concrete state into an abstract state and is therefore called an *abstraction function*.

SAT based predicate abstraction Most tools using predicate abstraction for verification use general-purpose theorem provers such as Simplify [23,**?**] to compute the abstraction. This approach suffers from the fact that errors caused by bit-vector overflow may remain undetected. Furthermore, bit-vector operators are usually treated by means of uninterpreted functions. Thus, properties that rely on these bit-vector operators cannot be verified. However, low-level software designs typically use an abundance of bit-vector operators, and that the property of interest will depend on these operations.

In [17], the authors propose to use a SAT solver, e.g., [41], to compute the abstraction of a sequential ANSI-C program. This approach supports all ANSI-C integer operators, including the bit-vector operators. We describe their technique below.

A transition relation $T(\bar{v}, \bar{v}')$ is computed for each statement (or basic block) in the given program. Let V be the set of variables in the given program. An assignment statement `v := exp` is transformed into an equality $v' = exp$. The primed version of a variable denotes the value of the variable in the next state (after executing the statement). This equality is conjoined with equalities that define the next value of any other variable $u \in V \backslash \{v\}$ to be the current value. Thus, only the value of the variable v in the assignment statement changes. This equation system is denoted by $\mathcal{T}$, $\overline{v}$ denotes the vector of all variables in V.

$$T(\overline{v}, \overline{v}') := v' = exp \wedge \bigwedge_{u \in V \backslash \{v\}} u' = u$$

Next a symbolic variable b_i is associated with each predicate π_i. Each concrete state $\bar{v} = \{v_1, \ldots, v_n\}$ maps to an abstract state $\bar{b} = \{b_1, \ldots, b_k\}$, where $b_i = \pi_i(\bar{v})$. If the concrete machine makes a transition from state $\bar{v}$ to state $\bar{v}' = \{v_1', \ldots, v_n'\}$, then the abstract machine makes a transition from state $\bar{b}$ to $\bar{b}' = \{b_1', \ldots, b_k'\}$, where $b_i' = \pi_i(\bar{v}')$.

The formula that is passed to the SAT solver directly follows from the definition of the abstract transition relation $\hat{T}$ as described in Section 2.1:

$$\hat{T} = \{(\bar{b}, \bar{b}') \mid \exists \bar{v}, \bar{v}' \ : \ \Gamma(\bar{v}, \bar{v}', \bar{b}, \bar{b}')\} \tag{1}$$

$$\Gamma(\bar{v}, \bar{v}', \bar{b}, \bar{b}') := \bigwedge_{i=1}^{k} b_i = \pi_i(\bar{v}) \ \wedge \ T(\bar{v}, \bar{v}') \ \wedge \ \bigwedge_{i=1}^{k} b_i' = \pi_i(\bar{v}') \tag{2}$$

The set of abstract transitions $\hat{T}$ is computed by transforming $\Gamma(\bar{v}, \bar{v}', \bar{b}, \bar{b}')$ into conjunctive normal form (CNF) and passing the resulting formula to a SAT solver. Suppose the SAT solver returns $\bar{v}, \bar{v}', \bar{b}, \bar{b}'$ as a satisfying assignment. We project out all variables but $\bar{b}$ and $\bar{b}'$ from this satisfying assignment to obtain an abstract transition $(\bar{b}, \bar{b}')$. Since we want all the abstract transitions, we add a blocking clause to the SAT equation that eliminates all satisfying assignments with the same values for $\bar{b}$ and $\bar{b}'$. This process is continued until the SAT formula becomes unsatisfiable. The satisfying assignments obtained form the abstract transition relation $\hat{T}$. As described in [14], there are numerous ways to optimize this computation. These techniques are beyond the scope of this article.

An abstract state $\bar{b}$ is an initial state in the abstract model, if there exists a concrete state $\bar{v}$ which is an initial state in the concrete model and maps to $\bar{b}$.

$$\hat{I} = \{\bar{b} \mid \exists \bar{v} : \ \bigwedge_{i=1}^{k} b_i = \pi_i(\bar{v}) \ \ \wedge \ I(\bar{v})\} \tag{3}$$

Using this definition, the abstract set of initial states can be enumerated by using a SAT solver.

4.2. Checking the abstract model

The abstraction process above results in a finite-state model, which can be checked using a finite-state modelchecker such as NuSMV [13]. If the abstract model satisfies the property, the property also holds on the original, concrete circuit. If model checking of the abstraction returns false, we obtain a counterexample from the model checker. In order to check if an abstract counterexample corresponds to a concrete counterexample, a *simulation* step is performed.

4.3. Simulation and Refinement

If the property does not hold on the abstract model, the model checker returns a counterexample trace. This trace is then checked on the concrete model. This process can be carried out using a theorem prover or a SAT solver as described below.

Let the counterexample trace have k steps. Each step in the abstract counterexample corresponds to a particular statement in the concrete program. The simulation requires a total of k SAT instances. Each instance adds constrains for one more step of the counterexample trace. We denote the value of the (concrete) variable $v \in V$ after step i by v_i. All the variables $v \in V$ inside an arbitrary expression e are renamed to v_i using the function $\rho_i(e)$.

The SAT instance number i is denoted by Σ_i and is built inductively as follows: Σ_0 (for the empty trace) is defined to be true. For $i \geq 1$, Σ_i depends on the type of statement of state i in the counterexample trace. Let p_i denote the statement executed in the step i.

If step i is a guarded goto statement, then the (concrete) guard g of the goto statement is renamed and used as conjunct. Furthermore, a conjunct is added that constraints the values of the variables to be equal to the previous values:

$$p_i = (\texttt{if } g \texttt{ then goto } l) \longrightarrow \Sigma_i := \Big(\Sigma_{i-1} \wedge \rho_i(g) \wedge \bigwedge_{u \in V} u_i = u_{i-1}\Big)$$

If step i is an assignment statement, the equality for the assignment statement is renamed and used as conjunct:

$$p_i = (\texttt{v:=exp}) \longrightarrow \Sigma_i := \Big(\Sigma_{i-1} \wedge \rho_i(v) = \rho_{i-1}(exp) \wedge \bigwedge_{u \in V \setminus \{v\}} u_i = u_{i-1}\Big)$$

Note that in case of assignment statement, Σ_i is satisfiable if the previous instance Σ_{i-1} is satisfiable. Thus, the check only has to be performed if the last statement is a guarded goto statement. If the last instance Σ_k is satisfiable, the simulation is successful and a bug is reported. The satisfying assignment provided by the SAT solver allows us to extract the values of all variables along the trace. If any SAT instance is unsatisfiable, the step number and the guard that caused the failure are passed to the refinement algorithm.

Refinement If the abstract counterexample cannot be simulated, it is an artifact from the abstraction process and the abstraction has to be refined. This is done by computing the weakest precondition of the guard g that caused the last SAT-instance Σ to be unsatisfiable. The weakest preconditions are computed following the simulation trace as built in the previous section. The new predicates obtained from these weakest pre-conditions are added to the global set of predicates.

4.4. Scalability

The application of predicate abstraction to large programs depends crucially on the choice and usage of the predicates. If all predicates are tracked globally in the program, the analysis often becomes intractable due to the large number of predicate relationships. In Microsoft's SLAM [3] toolkit, this problem is handled by generating coarse abstractions using techniques such as *Cartesian approximation* and the *maximum cube length approximation* [2]. These techniques limit the number of predicates in each theorem prover query. The refinement of the abstraction is carried out by adding new predicates. If no new predicates are found, the spu-

rious behavior is due to inexact predicate relationships. Such spurious behavior is removed by a separate refinement algorithm called Constrain [1].

The BLAST toolkit [32] introduced the notion of *lazy abstraction*, where the abstraction refinement is completely demand-driven to remove spurious behaviors. Recent work [30] describes a new refinement scheme based on interpolation [22,40], which adds new predicates to a selected set of program locations only. On average the number of predicates tracked at each program location is small and thus, the localization of predicates enables predicate abstraction to scale to larger software programs. Localization of predicates using weakest pre-conditions is described in [36].

5. Discussion and Conclusions

Although considerable progress has been made in development of techniques and tools for verification of programs, scalability of such approaches remains to be an issue. CEGAR based on predicate abstraction still requires an exponential cost in computing the abstraction as well inferring an optimal set of predicates for refinement. Pointers, recursive data structures and heaps are handled only to a limited extent in the current software verification tools. Finally, improved symbolic techniques for handling concurrency (*partial-order reduction* has been successful with explicit-state concurrent model checking) need to be developed.

We presented an overview of the current approaches to software verification together with details of a promising SAT-based CEGAR scheme using predicate abstraction. This technique allows precise treatment of ANSI-C constructs such as multiplication/division, pointers, bit-wise operations, type conversion and shift operators.

Acknowledgment. We would like to thank Constantinos Bartzis for his useful comments on this paper.

References

[1] T. Ball, B. Cook, S. Das, and S. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04*, pages 388–403. Springer, 2004.
[2] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS 01*, volume 2031, 2001.
[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The 8th International SPIN Workshop on Model Checking of Software, LNCS vol. 2057*, pages 103–122. Springer, May 2001.
[4] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
[5] Bandera website. `http://www.cis.ksu.edu/santos/bandera`.
[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. volume 1579, pages 193–207, March 1999.
[7] BLAST website. `http://www-cad.eecs.berkeley.edu/~rupak/blast`.

[8] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a java model checker. In *Workshop on Advances in Verification, Chicago, Illinois*, pages 130–135, 2000.

[9] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[10] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[11] S. Chaki, J. Ouaknine, K. Yorav, and Edmund M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of the 2nd Workshop on Software Model Checking (SoftMC '03)*, volume 89(3) of *Electonic Notes in Theoretical Computer Science*, July 2003.

[12] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2–3), 2004.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[14] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, Sep–Nov 2004.

[15] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[16] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[17] Edmund Clarke, Daniel Kroening, Natalia Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proc. of the Model Checking for Dependable Software-Intensive Systems Workshop, San-Francisco, USA*, 2003.

[18] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. volume 1855, pages 154–169, July 2000.

[19] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[20] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[21] M. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.

[22] William Craig. Linear reasoning. In *Journal of Symbolic Logic*, pages 22:250–268, 1957.

[23] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

[24] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. pages 177–187, May 2001.

[25] Matthew B. Dwyer, Robby, Xianghua Deng, and John Hatcliff. Space reductions for model checking quasi-cyclic systems. In *EMSOFT*, pages 173–189, 2003.

[26] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, October 2000.

[27] Patrice Godefroid. Software model checking: The Verisoft approach. *Formal Meth-*

ods in System Design, 2005. To Appear.
[28] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
[29] Aarti Gupta, Malay K. Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Abstraction and bdds complement sat-based bmc in diver. In *CAV*, pages 206–209, 2003.
[30] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, pages 232–244. ACM Press, 2004.
[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. volume 2725, pages 262–274, July 2003.
[32] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. volume 37(1), pages 58–70, January 2002.
[33] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
[34] Gerard J. Holzmann. Logic verification of ansi-c code with spin. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147, London, UK, 2000. Springer-Verlag.
[35] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Symposium on Leveraging Applications of Formal Methods*, 2004.
[36] Himanshu Jain, Franjo Ivančić, Aarti Gupta, and Malay K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, pages 397–412, 2005.
[37] Java PathFinder website. `http://javapathfinder.sourceforge.net`.
[38] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
[39] MAGIC. `http://www.cs.cmu.edu/~chaki/magic`.
[40] Kenneth L. McMillan. An interpolating theorem prover. In *TACAS*, pages 16–30, 2004.
[41] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.
[42] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS 99*, pages 178–192, 1999.
[43] SLAM website. `http://research.microsoft.com/slam`.
[44] Spin website. `http://spinroot.com`.