Predicate Abstraction of ANSI–C Programs using SAT

Edmund Clarke, Daniel Kroening, Natasha Sharygina and Karen Yorav
Carnegie Mellon University

Abstract. Predicate abstraction is a major method for verification of software. However, the generation of the abstract Boolean program from the set of predicates and the original program suffers from an exponential number of theorem prover calls as well as from soundness issues. This paper presents a novel technique that uses an efficient SAT solver for generating the abstract transition relations of ANSI-C programs. The SAT-based approach computes a more precise and safe abstraction compared to existing predicate abstraction techniques.

Keywords: Predicate Abstraction, ANSI-C, SAT

1. Introduction

It is widely believed that effective model checking [11] of software systems could produce major enhancements in software reliability and robustness. However, the effectiveness of model checking of such systems is severely constrained by the state space explosion problem, and much of the research in this area is targeted at reducing the state-space of the model used for verification. One principal method in state space reduction of software systems is *abstraction*. 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.

Predicate abstraction [20, 13] 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 abstract program is created using *Existential Abstraction* [10]. This method defines the transition relation of the abstract program so that it is guaranteed to be a *conservative* over-approximation of the original program, with respect to the set of given predicates. The use of a conservative abstraction, as opposed to an *exact* abstraction, produces considerable reductions in the state space. The drawback of the conservative abstraction is that when model checking of the abstract program fails it may produce a counterexample that does not correspond to a concrete counterexample. This is usually called a *spurious counterexample*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates this counterexample.

The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm [27, 2, 9, 17], or CEGAR for short. This framework is shown in Figure 1: 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. The steps are described below in the context of predicate abstraction.

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.

The efficiency of this process is dependent on the efficiency of the *program abstraction* and *predicate refinement* procedures. While program abstraction focuses on constructing the transition relation of the abstract program, the focus of predicate refinement is to define efficient techniques for choosing the set of predicates in a way that eliminates spurious counterexamples. In both areas of research low computational cost is a key factor since this is what enables the application of model checking to the verification of realistic programs.

In this paper we focus on the application of predicate abstraction to the verification of C programs. We present a novel technique that enables efficient

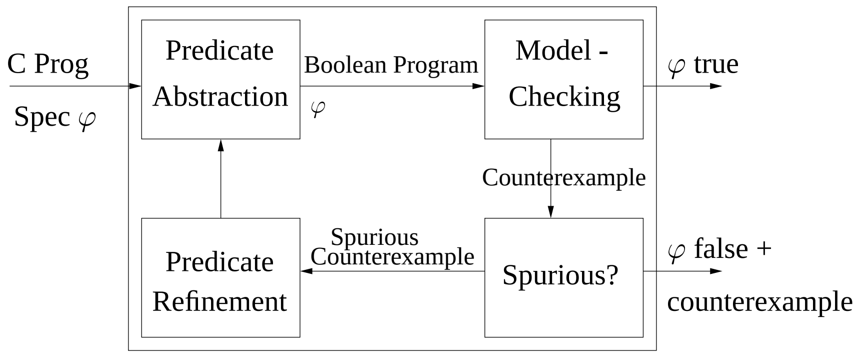

Figure 1. The Counterexample Guided Abstraction Refinement Framework

abstraction (Step 1 of the CEGAR loop) of a program by using a SAT solver to generate the abstract transition relation.

In previous work, including [3, 23, 5], the generation of the abstract Boolean program from a C program and a set of predicates suffers from multiple problems:

1. The generation of the Boolean program is done by calling a theorem prover for each potential assignment to the current and next state predicates. For the most precise transition relation, this requires an exponential number of calls of the theorem prover. Several heuristics are used to reduce this number. Some existing tools avoid this large number of theorem prover calls by using a user-specified maximum. After this specified number is reached, the tool adds all remaining transitions for which the theorem prover call was skipped. This is a safe over approximation, but will yield a potentially large number of unnecessary spurious counterexamples.

2. Existing work relies on general-purpose theorem provers. Program variables are modeled as unbounded integer values, neglecting possible arithmetic overflow in the ANSI-C program. This can result in false positive answers of the tool.

3. Existing tools only support a very limited range of operators, namely Boolean operators, addition/subtraction, equality, and relational operators. Other ANSI-C operators, such as multiplication and division, bitwise operators, type conversion operators, and shift operators are modeled by means of uninterpreted functions. This limits the set of programs and properties that can be verified.

4. Existing tools only provide a limited support for pointer operations. In particular, pointer arithmetic is not handled.

Contribution. This work proposes to use a SAT solver to generate the abstract program. The potentially exponential number of theorem prover calls is replaced by an enumeration on a single SAT instance.

For each basic block in the given program, our approach is to first construct a symbolic representation of the concrete transition relation by applying symbolic simulation techniques (similar to Currie et al. [15]). Next, we add the predicates in current and next state form to the relation between variables, resulting in a Boolean formula. Finally, we enumerate symbolically on the values of the predicates, using a SAT solver. When the abstract program needs to be refined, we use the same formula that we have already created, together with the new set of predicates, to create the new abstraction.

The advantage of this technique is that the exponential number of theorem prover calls is eliminated; instead, the possible assignments to the values of the predicates are searched by the SAT solver. Modern SAT solvers are highly efficient, and allow a large number of variables. This enables checking many more possible assignments, resulting in a more precise abstract transition relation, and eliminating redundant spurious counterexamples.

Another advantage of our approach is that most ANSI-C constructs can be encoded using CNF, which allows a wider range of programs. Integer operators are encoded using bit vector operators, i.e., they take into account the potential arithmetic overflow. Thus, there are no false positive answers due to the inaccurate assumption that the range of values of the variables is infinite. Moreover, pointer manipulation constructs, including pointer arithmetic, can also be supported. The only limitation is that recursion and dynamic memory allocation are not allowed. This limitation cannot be avoided, since the Boolean program is required to be finite. The symbolic simulation technique we use is taken from Kroening et al. [25].

Related Work. Data abstraction techniques are widely used and they have been explored by a large number of researchers [10, 16, 27, 29, 24]. Abstraction techniques are often based on the abstract interpretation work of Cousot and Cousot [14] and require the user to give an abstraction function relating concrete datatypes to abstract datatypes. Earlier applications of the predicate abstraction type of the abstract interpretation approach [20, 4, 13] require the user to identify the set of predicates that influence the verification property and utilize general-purpose theorem proving to compute the abstract program. The user-driven discovery of relevant predicates makes these methods less effective for large programs.

Recently, various decision procedures have been proposed to compute the set of predicates for abstraction. The most common approach is to use error traces [9, 1] to guide the predicate discovery. In Clarke et al. [9], the algorithm is based on BDD representations of the program. This is a drawback for large programs, where transition relation BDDs are commonly too large for efficient manipulation. The algorithm presented in the work of Ball et al. [1] uses an explicit state representation but it is restricted to safety properties.

The abstraction refinement loop was first introduced by Kurshan [27]. The localization reduction technique defined in [27] produces an initial abstraction of the program by ”freeing away” program variables that do not affect the verification property. The values of “free” variables are defined nondeterministically, which results in an over-approximation of the program behaviors. The unrealistic behaviors are eliminated from the program by gradually refining the “free” variables back to their original values.

Clarke et al. [9] extended the work of Kurshan [27] by defining a procedure for systematic abstraction refinement. Spurious error traces are used by the refinement decision procedure in order to ensure that the new abstraction will not allow this counterexample.

Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement has become a widely applied technique. It was introduced by Ball and Rajamani [2] and promoted by the success of the SLAM project [3]. The goal of this project is to verify that Windows device drivers obey API conventions. SLAM models program variables using unbounded integer numbers, and does not take overflow or bit-wise operators into account. The abstraction of the program is computed using a theorem prover such as Simplify [18]. The property checked mainly depends on the control flow, and thus, this treatment is sufficient. However, there are C programs make extensive use of bit-wise operators, for example programs that represent a circuit model. For these programs we expect that the limited range of the variables will be crucial. BOOP [5] is a re-implementation of SLAM. BLAST is another software model checker for C programs that uses the counterexample-driven automatic abstraction refinement to construct an abstract program. The abstraction is constructed on-the-fly and only to the required precision [23].

The NASA Ames model checker, JavaPathFinder [6], developed for verifying Java programs, was also reported to use heuristics for automated predicate abstraction and refinement. In this tool, predicate abstraction procedures are extended with some informal abstraction arguments that allow predicate abstraction to be used within the class-instance of object-oriented languages. The CMU concurrent C model checker, MAGIC [7], applies automatic compositional reasoning on programs with functions.

Recently, there has been some work reported on the application of SAT solvers in the process of constructing and refining predicate abstractions. Previously, the application of SAT solvers during computation of predicate abstraction was conducted only in the context of hardware verification in the work of Clarke et al. [12]. The focus of [12], indeed, is the refinement of the initial approximate abstraction, and not the construction of the abstraction itself. The approximate abstract model is constructed by excluding certain implications from consideration. In contrast, we use a SAT solver to construct the exact abstract transition relation according to the provided set of predicates, rather than an approximation of it.

Strichman et al. [8] use a SAT engine for identifying (or approximating) the minimal set of predicates needed to eliminate a set of spurious counterexamples during refinement of abstract C programs. The predicate minimizing algorithm is implemented in the MAGIC tool, which uses a theorem prover to compute predicate abstraction.

To our knowledge, the technique reported in this paper is the first effort to apply a SAT engine for the actual construction of a predicate abstraction of software. The reported technique is defined in the context of ANSI-C programs. However, the method is general and can be applied to programs written in other imperative programming languages.

The article is structured as follows: Section 2 discusses the details of constructing a Boolean formula for the concrete transition relation. Section 3 describes how a SAT solver is used to compute the abstraction. Section 4 gives some details about the implementation of our ideas, and presents some experimental results. Finally, Section 6 summarizes the contributions of the article.

2. A Boolean Formula for the Concrete Transition Relation

This Section discusses the details of constructing a Boolean formula for the concrete transition relation. The program is first partitioned into basic blocks, which are sequentially composed assignments, and control flow statements, i.e., `if`, `while`, `goto` and so on.

We use bit-vector equations to capture the semantics of assignments. This implies a different approach for control-flow statements and basic blocks. Since control-flow statements do not change variable values (we remove side-effects from conditions in a pre-processing step), they do not require equations. The abstraction of control statements is therefore not described here, but is deferred to Section 3.2. For the rest of this section we are only concerned with basic blocks.

Section 2.1 describes syntactic program transformations that are required to prepare the basic block for the translation into a bit-vector equation. Section 2.2 gives details on how assignments are translated into bit-vector equations using symbolic simulation techniques. Section 2.3 presents details how this is done in the presence of pointers. The translation described here is an adaptation of the method presented in [25, 26]. Section 2.4 shows the translation of the generated bit-vector equation system into a Boolean formula, which is suitable for a SAT solver.

2.1. Preparation

For the rest of this section we assume B is a basic block containing n statements $s_1, \ldots, s_n$. This code has already been manipulated to remove function calls and empty (skip) statements, so we can assume that each s_i is an assignment. We use the notation $\mathit{lhs}(s_i)$ and $\mathit{rhs}(s_i)$ for the left-hand side and right-hand side of the assignment, respectively.

Given an expression e, we use $\mathit{Vars}(e)$ to denote the set of variables referenced by this expression. We use this notation also for assignments, so that $\mathit{Vars}(s_i) = \mathit{Vars}(\mathit{lhs}(s_i)) \cup \mathit{Vars}(\mathit{rhs}(s_i))$.

We first transform B into single assignment form, in which each variable is assigned to only once. In order to do so, we add auxiliary variables that record intermediate values. Let v be a variable and s_i an assignment such that $v \in \mathit{Vars}(s_i)$. Let $\alpha(v, s_i)$ denote the number of assignments made to variable v within the basic block prior to the statement s_i. Formally,

$$\begin{aligned} \alpha(v, s_1) &= 0 \\ \forall i \geq 2 : \alpha(v, s_i) &= \begin{cases} \alpha(v, s_{i-1}) + 1 & : s_{i-1} \text{ assigns to } v \\ \alpha(v, s_{i-1}) & : \text{otherwise} \end{cases} \end{aligned}$$

DEFINITION 1 (ρ). *Let s_i be an assignment that assigns to the variable v. Then the leftmost occurrence of v in $\mathit{lhs}(s_i)$ is renamed to $v_{\alpha(v,s_i)+1}$. All other occurrences of v are renamed $v_{\alpha(v,s_i)}$. Any other variable $u \in \mathit{Vars}(s_i)$ such that $u \neq v$ is renamed $u_{\alpha(u,s_i)}$.*

Let e denote any expression (whether a part of an assignment, a whole assignment, a condition, etc). Then $\rho(e)$ denotes the expression after this renaming.

Figure 2 gives an example of a simple block and its translation.

```
x = z * x;
y = x + 1;
x = x + y;
```

$\xrightarrow{\rho}$

$$\boxed{\begin{array}{l} \mathtt{x}_1 \mathtt{\ =\ z}_0 \mathtt{\ *\ x}_0\mathtt{;} \\ \mathtt{y}_1 \mathtt{\ =\ x}_1 \mathtt{\ +\ 1;} \\ \mathtt{x}_2 \mathtt{\ =\ x}_1 \mathtt{\ +\ y}_1\mathtt{;} \end{array}}$$

Figure 2. Translation of a basic block into its single assignment form.

In the following, we use v for a program variable (such as `x` in the example above) and v_j for one of its renamed versions ($\mathtt{x}_0$, $\mathtt{x}_1$, $\mathtt{x}_2$ in that example).

2.2. Translating assignments into bit-vector equations

We next define an equation $\sigma(s_i)$ for each assignment in the block, describing the effect this assignment has on the (renamed) variables. In this subsection we assume that the program does not have any pointer variables; sub-section 2.3 will extend the method to programs that manipulate pointers.

As an intermediate format, we use bit-vector equations. Besides the usual bit-wise and arithmetic operators, we also consider the array index operator [ ], the structure member operator, and the choice operator to be part of the logic. The choice operator "?" is defined as:

$$c?a : b \ \stackrel{\triangle}{=} \ \begin{cases} a & : \ c \neq 0 \\ b & : \ \text{otherwise} \end{cases}$$

Furthermore, we define the `with` operator [21] for arrays and structures. It is also considered part of the bit-vector logic.

DEFINITION 2 (`with` operator for arrays). *Let g be an expression of array type, i be an integer expression, and e be an expression with the type of the elements in g. The operator* `with` *takes g, i, and e and produces an array that is identical to g, except for the content of $g[i]$ being replaced by e. Formally, let g' be "g* `with` *$[i] := e$", then*

$$g'[j] \ \stackrel{\triangle}{=} \ \begin{cases} e & : \ j = i \\ g[j] & : \ \textit{otherwise} \end{cases}$$

DEFINITION 3 (`with` operator for structures). *Let s be a variable of structure type, f be a field name of this structure, and e be an expression matching the type of the field f. The operator* `with` *takes s, f, and e and produces a structure that is identical to s, except for the content of $a.f$ being replaced by e. Formally, let s' be "s* `with` *$.f := e$" and j be a field name of s, then*

$$s'.j \ \stackrel{\triangle}{=} \ \begin{cases} e & : \ j = f \\ s.j & : \ \textit{otherwise} \end{cases}$$

The translation of an assignment into a constraint is done using an auxiliary function $\ell(l, r)$. It maps the expressions l for the left hand side and r for the right hand side into a constraint. It is defined recursively on the structure of the expression l:

– If l is a symbol v, then $\ell(l, r)$ is the equality of the left hand side l and the right hand side r.

$$\ell(v, r) \ := \ v = r$$

– If l is an array index expression $g[i]$ with array expression g and index expression i, then $\ell(l, r)$ is applied recursively to g and a new right hand side which is g with element i changed to r.

$$\ell(g[i], r) \;:=\; \ell(g, g \;\mathsf{with}\; [i] := r)$$

– If l is a structure member expression $s.f$ with structure expression s and field name f, we define $\ell(l, r)$ in analogy to the previous case:

$$\ell(s.f, r) \;:=\; \ell(s, s \;\mathsf{with}\; .f := r)$$

Using this auxiliary function, the function $\sigma(s_i)$ is easily defined as

$$\sigma(s_i) \;:=\; \ell(lhs(s_i), rhs(s_i))$$

Our final bit-vector equation is the conjunction of the constraints generated:

$$\bigwedge_{i=1,\ldots,n} \sigma(s_i)$$

As a shorthand, let v denote the version of the variable v with index 0, and v' denote the version of the variable v with the largest index, or formally

$$\begin{aligned} v &:= v_0 \\ v' &:= v_{\alpha(v, s_{n+1})} \end{aligned}$$

Note that for any variable v that is not assigned to within the basic block, v' is just another shorthand for v_0. This gives us a bit-vector equation system that defines a relation $\mathcal{T}(\overline{v}, \overline{v}')$, where $\overline{v}$ is the vector of all variables v, and $\overline{v}'$ is the vector of all variables v'. The relation is a symbolic representation of the concrete transition relation of the block B, i.e., the vector $\overline{v}$ represents the state before the execution of the basic block, and $\overline{v'}$ represents the state after the execution of the basic block. Every solution to this equation system represents a possible computation of the basic block.

2.3. Programs that use pointers

While other tools rely solely on static analysis techniques to determine the set of variables a pointer may point to, we also look at the *predicates*. As will be evident in the following, the size of the generated equation for a statement involving a pointer p is linear in the number of objects p may point to. Thus, it is desirable to keep this number small. In a typical application there may be a large number of variables having the correct type as $*p$, while only a few that p can actually point to. In order to minimize the size of the equation generated we use all the information we can extract from the program about the possible targets of p. Using the (dynamic) information obtained from the predicates, we can save a lot more than by merely using static points-to algorithms.

Before giving the formal definition, we motivate our construction as follows: When a pointer p is dereferenced and the abstract state does not hold enough information to guarantee that p is a valid, active object, the abstract program must generate an exception. This is necessary to make the abstraction safe, i.e., the abstract program can refrain from generating an exception only if it is guaranteed that the concrete program does not generate an exception. For example, assume that p may point to one of $\{x, y, z\}$, while the set of predicates that involve p is $\{(p = \&x), (p = \&y)\}$. The abstract program cannot distinguish between p pointing to z, or p being `NULL`, or even p pointing to some other illegal address. Whenever p is dereferenced while both predicates are false, the abstract program will generate an exception. This means that when creating the abstract transition relation we can ignore the possibility of p pointing to z, and treat it in the same way as if p were `NULL`.

The concrete transition relation we generate therefore actually depends on the predicates, and is already an abstraction of the concrete behavior.

Let $\Theta(p)$ denote the set of variables which p can legally point to (i.e., the variables with a compatible type). The variables in $\Theta(p)$ are variable names *before* renaming. We analyze the set of predicates $\mathcal{P}$ and extract a set of variables $\theta(p, \mathcal{P}) \subseteq \Theta(p)$, such that $v \in \theta(p, \mathcal{P})$ holds if it is possible to derive from the predicates that p points to v. This information usually comes from predicates of the form $p = \&x$, $p = \&x + i$, $p = q$, and so on.

DEFINITION 4 ($\theta(p, \mathcal{P})$). *Let $\mathcal{P} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates. Then $\theta(p, \mathcal{P})$ is the set of variables $v \in \Theta(p)$ for which there exists a truth assignment to the predicates such that the resulting conjunction implies that p holds the address of v.*

$$\theta(p, \mathcal{P}) \overset{\triangle}{=} \{v \in \Theta(p) \mid \exists b_1, \ldots, b_k.(\bigwedge_{i=1,\ldots,k} (b_i \leftrightarrow \pi_i) \Rightarrow (p = \&v)\}$$

A pointer dereference `*p` in an expression is replaced by a case split on all the variables from $\theta(p, \mathcal{P})$. Let $\theta(p, \mathcal{P}) = \{v^1, \ldots, v^k\}$. We replace every occurrence of `*p` with

(p==&v^1) ? v^1 : (p==&v^2) ? v^2 : ...(p==&v^k) ? v^k : $\perp$

where $\perp$ is a default value, which is never used. It is important to notice that &v^i (the address of v) is a constant value and does not get renamed, while v^i is a variable name and will be given an index during the renaming process ρ. The end result is that when a pointer is dereferenced in the right-hand side of an assignment, or in the index of an array on the left-hand side, the correct value

```
x = 5;
y = *p +
1;
*p = 2*y;
```

$$\xrightarrow{\rho}\quad \boxed{\begin{array}{l}\texttt{x}_1\texttt{ = 5;}\\ \texttt{y}_1\texttt{ = ((p}_0\texttt{==\&x) ? }x_1\texttt{ : y}_0\texttt{) + 1;}\\ \texttt{x}_2\texttt{ = (p}_0\texttt{==\&x) ? 2*}y_1\texttt{ : x}_1\texttt{;}\\ \texttt{y}_2\texttt{ = (p}_0\texttt{==\&y) ? 2*}y_1\texttt{ : y}_1\texttt{;}\end{array}}$$

$$\xrightarrow{\sigma}\quad \boxed{\begin{array}{c} x_1 = 5 \quad \wedge \\ y_1 = ((p_0 = \&x)\,?\,x_1\,:\,y_0) + 1 \quad \wedge \\ x_2 = (p_0 = \&x)\,?\,2 * y_1\,:\,x_1 \quad \wedge \\ y_2 = (p_0 = \&y)\,?\,2 * y_1\,:\,y_1 \end{array}}$$

Figure 3. Example: Generation of the concrete transition relation. As an optimization, we restrict the case splits done for pointers using information from the predicates. For this example, assume the predicates $p = \&x$ and $p = \&y$.

will be used. Note that it is not necessary to include all variables in $\Theta(p)$, since we generate an exception if p does not point to an object in $\theta(p, \mathcal{P})$.

The case where a pointer dereference appears on the left hand side of an assignment is again handled by a transformation of the program, before renaming is applied. The assignment `*p = exp` is capable of affecting any variable with the correct type. We therefore replace this assignment with a series of assignments. For each variable $u \in \theta(p, \mathcal{P})$, we add an assignment of the following form:

$$u\ \texttt{= (p==\&}u\texttt{) ? exp : }u\texttt{;}$$

Again, we may refrain from adding an assignment for any variable not in $\theta(p, \mathcal{P})$ since if p points to such a variable there will be an exception.

The transformed program does not have pointer dereferences, and can be translated into an equation system using the σ function presented in the previous section. Notice that for the assignment $p = \&x$ the rule for $\sigma(v_j = exp)$ applies without change. The address of a variable is treated as a value and is assigned into a variable with an appropriate type.

An example of the process described above is given in Figure 3. The example gives a basic block, the renamed version, and the resulting equation system.

2.4. Translating bit-vector equations into Boolean formulas

The translation of the bit-vector logic used to build the equation for the concrete transition relation is straightforward: we build a circuit representation, which is then translated into CNF. Several optimizations can be done at this level, in particular for arrays. The result of this process is a CNF formula $\mathcal{T}(\overline{v}, \overline{v}')$ that is a symbolic representation of the concrete transition relation.

3. Using SAT to Compute the Abstraction

3.1. The abstract transition relation for a basic block

Let $\mathcal{P}$ be the set of predicates, where each predicate is an expression over the (concrete) program variables. Each predicate $\pi_i \in \mathcal{P}$ is associated with a Boolean variable b_i that represents its truth value. Let $\overline{\pi}$ denote the vector of predicates π_i, and $\overline{b}$ denote the vector of the Boolean variables b_i. These Boolean variables are the variables of the Boolean program we are constructing. The predicates map a concrete state $\overline{v}$ into an abstract state $\overline{b}$, and thus, $\overline{\pi}(\overline{v})$ is also called the abstraction function. Given $\mathcal{T}(\overline{v}, \overline{v}')$ and $\mathcal{P}$, we create an abstract transition relation $\mathcal{B}(\overline{b}, \overline{b}')$ that is an existential abstraction of a basic block of the C program.

Our goal is to replace a basic block with an expression that describes what happens to the variables $\overline{b}$ when this basic block is executed. We present a translation that is accurate, i.e., it gives the transition relation as defined by existential abstraction, and not an over-approximation of this transition relation, as other tools use.

Let $\mathcal{T}(\overline{v}, \overline{v}')$ denote the CNF formula representing the concrete transition relation, as defined in the previous section. The abstract transition relation $\mathcal{B}(\overline{b}, \overline{b}')$ relates a current state $\overline{b}$ (before the execution of the basic block) to a next state $\overline{b}'$ (after the execution of the basic block). It is defined using $\overline{\pi}$ as follows:

$$\Gamma(\overline{b}, \overline{b}', \overline{v}, \overline{v}') \quad \stackrel{\triangle}{=} \quad (\overline{\pi}(\overline{v}) = \overline{b}) \wedge \mathcal{T}(\overline{v}, \overline{v}') \wedge (\overline{\pi}(\overline{v}') = \overline{b}') \tag{1}$$

$$\mathcal{B}(\overline{b}, \overline{b}') \iff \exists \overline{v}, \overline{v}' : \Gamma(\overline{b}, \overline{b}', \overline{v}, \overline{v}') \tag{2}$$

The concrete transition relation $\mathcal{T}$ maps a concrete state $\overline{v}$ into a concrete next state $\overline{v}'$, and the abstract transition relation $\mathcal{B}$ maps a corresponding abstract state $\overline{b}$ into a corresponding abstract next state $\overline{b}'$. The abstraction function $\overline{\pi}$ maps the concrete states into abstract states. Put together, we get the classical abstraction connection:

$$\begin{tikzcd}
\overline{v} \arrow[r, "\mathcal{T}(\overline{v}, \overline{v}')"] \arrow[d, "\overline{\pi}"'] & \overline{v}' \arrow[d, "\overline{\pi}"] \\
\overline{b} \arrow[r, "\mathcal{B}(\overline{b}, \overline{b}')"'] & \overline{b}'
\end{tikzcd}$$

Every satisfying assignment to (1) represents a concrete transition and its corresponding abstract transition. We aim at obtaining all possible satisfying assignments to the abstract variables $\overline{b}$ and $\overline{b}'$, i.e., the set

$$\{(\overline{b}, \overline{b}') \mid \mathcal{B}(\overline{b}, \overline{b}')\} \tag{3}$$

This set is obtained by modifying the SAT solver Chaff as follows: every time a satisfying assignment is found, the tool records the values of the literals corresponding to the abstract variables $\overline{b}$ and $\overline{b}'$, and then adds a blocking clause in terms of these literals that eliminates all satisfying assignments where these variables have the newly found values. The literals in the blocking clauses all have a decision level, since the assignment is complete. The solver then backtracks to the highest of these decision levels and continues its search for further, different satisfying assignments. Thus, the SAT solver is used to enumerate the set (3). This technique is commonly used in other areas, for example in [30, 22]. Section 4 contains more details on how to efficiently obtain the set of satisfying assignments.

As an example, consider the following basic block:

```
d=e;
e++;
```

where `d` and `e` are integer variables. Suppose the predicates $\pi_1 = d\&1$ and $\pi_2 = e\&1$ are given. The binary operator $\&$ is the bit-wise conjunction operator, i.e., π_1 holds if and only if `d` is odd, and π_2 holds if and only if `e` is odd. The basic block is translated into the following equation system, which represents the transition relation:

$$d_1 = e_0 \wedge e_1 = e_0 + 1 \tag{4}$$

By adding the required constraints according to equation (2) we get:

$$\begin{aligned} b_1 = d_0\&1 \wedge b_2 = e_0\&1 \wedge \\ d_1 = e_0 \wedge e_1 = e_0 + 1 \wedge \\ b_1' = d_1\&1 \wedge b_2' = e_1\&1 \end{aligned} \tag{5}$$

The satisfying assignments for this equation over the variables b_1, b_1', b_2, and b_2' are:

b_1	b_2	b_1'	b_2'
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

In particular, the abstract Boolean program will never make a transition into a state that is contradictory in the sense that both `d` and `e` (which is equal to `d + 1`) are odd. This is unavoidable if a next state function is computed separately for each Boolean variable b_i, as done by many existing tools.

Consider the basic block above with the predicates $\pi_1 = e \geq 0$ and $\pi_2 = e \leq 100$, and suppose that `x` has 32 bits. The equation for the abstract transition relation $\mathcal{B}$ is:

$$\begin{gathered} b_1 = e_0 \geq 0 \wedge b_2 = e_0 \leq 100 \wedge \\ d_1 = e_0 \wedge e_1 = e_0 + 1 \wedge \\ b'_1 = e_1 \geq 0 \wedge b'_2 = e_1 \leq 100 \end{gathered} \tag{6}$$

The satisfying assignments for this equation over the variables b_1, b'_1, b_2, and b'_2 are:

b_1	b_2	b'_1	b'_2
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	1	0
1	1	1	1

Note that incrementing a positive number is not guaranteed to yield another positive number because of the finite range (there is a transition from a state with $b_1 = 1$ to a state with $b'_1 = 0$).

3.2. The abstract transition relation for control-flow statements

Besides basic blocks, the concrete program also contains control flow statements such as `if` and `while`. These statements take a condition as an argument and affect only the control-flow (the program counter). We pre-process the program to remove all side-effects from conditions. Since control-flow statements do not change the values of variables, we do not require an equation system to represent them.

Assume we are abstracting a specific program counter location l that evaluates a condition c and moves the program counter to location l_T if c holds and l_F otherwise. Our goal is to generate two sets of abstract transitions, a set of transitions that assign l_T to the program counter, and a set that assigns l_F. All of the transitions will leave the abstract variables $\overline{b}$ unchanged.

To abstract c we first traverse its syntactic structure to see whether there are any sub-expressions that are also predicates in $\mathcal{P}$. We replace any occurrence of a predicate π_i in c with the corresponding Boolean variable b_i. Let c_1 be the condition that results from applying this transformation. If c_1 references only

Boolean variables then we are done – this condition can be used in the abstract program. We then generate an abstract statement that assigns the program counter with l_T if $c_1(\overline{b})$ holds, and l_F otherwise.

If, however, c_1 still refers to some concrete variables $\overline{v}$, we use the SAT enumeration engine in order to produce the set of abstract transitions that correspond to the evaluation of c.

The condition $c(\overline{v})$ holds in an abstract state $\overline{b}$ if and only if there is a concrete state $\overline{v}$ such that the condition holds in $\overline{v}$ and $\overline{v}$ is mapped to $\overline{b}$. To create the abstract transition relation at this control location we need to produce the set pos_c of abstract states from which there is a transition that assigns the program counter with l_T:

$$pos_c = \{\overline{b} \,|\, \exists \overline{v} : c(\overline{v}) \wedge \overline{\pi}(\overline{v}) = \overline{b}\} \tag{7}$$

The dual set neg_c of abstract states from which there is a transition that assigns the program counter with l_F is *not* the negation of pos_c. This is because a single abstract state can correspond to both concrete states that satisfy c and concrete states that do not. We are therefore required to generate the set neg_c according to its definition:

$$neg_c = \{\overline{b} \,|\, \exists \overline{v} : \neg c(\overline{v}) \wedge \overline{\pi}(\overline{v}) = \overline{b}\} \tag{8}$$

Both of these sets are computed using the SAT enumeration engine.

In practice, we are rarely required to use the SAT enumeration engine for control-flow statements. The conditions of `if` statements and `while` loops are often chosen as Boolean predicates. Furthermore, most refinement algorithms will add these conditions whenever they encounter a spurious counterexample that passes through this statement.

4. The Implementation

4.1. Minimizing the Number of Quantified Variables

The size of the set (3) described in the previous section can be exponential in the number of predicates. However, in practice, a basic block usually mentions a very small subset of all program variables. Thus, most Boolean program variables are usually unchanged in the abstract version of the basic block. In particular, if a predicate uses only variables that are not assigned to, the truth value of the predicate is guaranteed not to change. We call the remaining predicates (the predicates that use variables that get assigned into) the *output predicates*. Formally, these are the predicates π_i such that $\pi_i(v) \neq \pi_i(v')$.

Furthermore, we try to detect which predicates can actually influence the next abstract values of the output predicates. This is done by obtaining the set of variables that are used in the assignments to variables that are mentioned in output predicates. We call these predicates the *input predicates*.

Example. As an example, consider the predicates $\pi_1 = \texttt{i} > \texttt{10}$ and $\pi_2 = \texttt{j} > \texttt{10}$. Let the basic block consist only of the statement

```
i=j;
```

In this case, π_1 is the only output predicate (as `j` is not modified) and π_2 is the only input predicate (as `i` is not mentioned in the right hand side).

As an optimization, we only obtain the projection of the set (3) on the input and output predicates, where $\overline{b}$ is restricted to contain only input predicates and $\overline{b}'$ is restricted to only contain output predicates.

4.2. Obtaining the Set of Satisfying Assignments

The problem of obtaining the set of satisfying assignments to a formula restricted to a given subset of the variables corresponds to a quantification problem. Let S denote the subset of variables. We obtain the set by enumeration on the variables in S using a SAT solver. This method was suggested earlier for solving quantified formulae in [32, 33]. In [28], our implementation algorithm was applied to predicate abstraction for hardware and software systems. It outperformed BDDs on all software examples. These results were obtained using arithmetic on integers however, not on bit-vectors.

The basic algorithm works as follows: when the SAT solver finds a satisfying assignment, it generates a blocking clause in terms of the variables in S. This blocking clause is added to the clause data base and prohibits any further satisfying assignment with the same values for the variables in S. After adding the clause to the CNF, the algorithm performs backtracking to the highest decision level of any of the variables in the blocking clause and continues the search for more satisfying assignments. Eventually, the additional constraints will make the problem unsatisfiable, and the algorithm terminates. The blocking clauses added by the algorithm are a DNF representation of the desired set.

Each DNF clause represents a hyper-cube, and is contained in the set of solutions. The basic algorithm can be improved by heuristics that try to enlarge the cube represented by each clause. In [30], McMillan uses conflict graph analysis in order to enlarge the cube. In [22], BDDs are used for the enlargement. However, these techniques are beyond the scope of this article.

4.3. Using SMV to Check the Abstract Program

We use the CMU version of SMV [34] to verify the abstract program. The advantage of using SMV is that the hyper-cubes representing the abstract transition relation can be passed to SMV directly by means of the `TRANS` statement. However, any other unbounded model checker is applicable as well, including SAT-based model checkers.

The control flow of the abstract program (which matches the control flow of the concrete program) is realized by adding a program counter variable. Each control flow location corresponds to a set of hyper-cubes.

For the second example in section 3, we obtain four cubes representing the six satisfying assignments:

$$\begin{array}{cccccccc} & \neg b_1 & \wedge & b_2 & \wedge & \neg b_1' & \wedge & b_2' \\ \vee & b_1 & \wedge & \neg b_2 & \wedge & \neg b_1' & \wedge & b_2' \\ \vee & b_1 & & & \wedge & b_1' & \wedge & \neg b_2' \\ \vee & & & b_2 & \wedge & b_1' & \wedge & b_2' \end{array}$$

Assuming the PC of this statement is `x`, this corresponds to the following `TRANS` statement:

```
TRANS PC=x -> (!b1 &  b2 & !next(b1) &  next(b2))
            | ( b1 & !b2 & !next(b1) &  next(b2))
            | ( b1       &  next(b1) & !next(b2))
            | (       b2 &  next(b1) &  next(b2))
```

4.4. Simulating the Abstract Counterexample

If the Model-Checker detects that the property does not hold on the abstract program, it generates a counterexample trace. This trace is then simulated on the concrete program in order to determine whether the counterexample is spurious or not. Most existing tools use a theorem prover such as Simplify [18] for this task.

The disadvantage of using a general purpose theorem prover for the simulation of the counterexample are similar to the disadvantages that arise during the computation of the abstract transition relation: The set of operators is limited, and the theorem prover may misjudge a counterexample to be real due to the lack of overflow detection.

The methodology that is used to obtain the concrete transition relation is also applicable to simulate the counterexample: Following the control flow in the abstract trace, we concatenate the corresponding basic blocks of the concrete program and apply the symbolic simulation technique described earlier.

We then incrementally add the constraints that the control flow in the abstract trace impose, i.e., the concretized versions of the control flow conditions. After adding a new control flow condition as a constraint, we check the satisfiability of the equation using SAT. If the equation is satisfiable, the abstract trace can be simulated so far. If it is unsatisfiable, the abstract trace cannot be simulated and is therefore spurious.

If all control flow conditions found in the abstract trace are added and the equation is still satisfiable, the abstract trace can be simulated on the concrete program, and thus, a bug has been found. The tool then prints out the concrete trace. The values of the concrete variables can be obtained directly from the satisfying assignment.

In comparison to the concrete program, the control flow conditions are small. Thus only a few clauses and variables are added to the CNF in each step. We therefore use an incremental SAT solver [31] in order to preserve the information learned by the solver between the satisfiability checks.

4.5. Verifying properties of the program

The setup described so far can be used to check reachability of code locations, as done by other tools such as SLAM, BLAST or BOOP. In addition to that, we check several safety properties such as array bounds and user defined assertions.

The ANSI-C standard stipulates that at any point in the program one can insert an `assert` statement that specifies a Boolean condition. For example, the program

```
x = y;
y = y + 1;
assert(y > x);
```

asserts that after the two assignments `y` will be greater than `x`. This assertion fails if incrementing `y` results in an overflow. Assertions are placed in the program as a specification of correctness. In order to verify the program we must determine that the condition in the assertion is true in all possible executions.

When creating the abstract program we translate every `assert(C)` statement, where `C` is a Boolean condition, by abstracting the condition `C`. This is done using the same method that we use for the conditions of "if" and "while" statements, as described in section 3.2.

In addition to user specified assertions we verify several basic correctness properties of the program.

– Whenever a basic block contains a dereference $*p$ of a pointer variable p, we check that the pointer cannot be pointing to an illegal address. Let $\theta(p, \mathcal{P})$ be the set of variables for which the predicates can imply that p is pointing to (Definition 4). We then check that

$$\bigvee_{v \in \theta(p,\mathcal{P})} (p = \&v)$$

is valid by abstracting the expression as as described in section 3.2.

– Whenever a basic block contains a reference to an element of an array we make sure that the array boundaries are not violated. If the expression $a[i]$ appears in the basic block (where i may be any integer expression), and the array a is of length n, we check

$$(i < n) \wedge (i \geq 0)$$

for validity.

– Whenever the basic block contains an expression that performs division, i.e., an expression of the form x/y (where y can be any numeric expression) we make sure that the divisor is not zero.

5. Experimental Results

We applied the SAT-based abstraction approach to the verification of several C programs.

5.1. SHA

We used a program taken from the Digital Signature Standard (DSS). Under the DSS, communication among remote parties is enabled using digital signatures. The digital signature is computed using two inputs: 1) a delivery message of the communication instance; and 2) a private key of a public/private key pair. We verified the C implementation of the DSS Secure Hash Algorithm (SHA) [19].

The SHA computes a part of the DSS digital signature called the message digest. The hashing algorithm computes the message digest by generating a 160-bit representation of the delivery message. The hashing procedure is designed to assure that the digest is statistically unique. The implementation makes extensive use of bit-wise operators and also division.

The code contains calls to `abort()` in places where an unexpected condition happens, e.g., an arithmetic error. These calls can be considered an implicit property. We replace these calls by `assert(0)`, i.e., we prove that these program locations are not reachable. The reachability of one of these locations depends on the result of a division: the code divides a 32-bit variable `t` by 20, and then checks that the result is between 0 and 3 using a `switch` statement. If the result is any other value (default case), `abort()` is called (figure 4). The property holds as the range of `t` is limited appropriately.

Given one predicate for each of the four possible `switch` cases, our tool generates an abstract transition relation that is consistent (at most one of the four, mutually exclusive predicates holds) and strong enough to show

```
switch ( t / 20 )
  {
   case 0:
      TEMP2 = ( (B AND C) OR (˜B AND D) );
      TEMP3 = ( K_1 );
      break;

   case 1:
      TEMP2 = ( (B XOR C XOR D) );
      TEMP3 = ( K_2 );
      break;

   case 2:
      TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
      TEMP3 = ( K_3 );
      break;

   case 3:
      TEMP2 = ( B XOR C XOR D );
      TEMP3 = ( K_4 );
      break;

   default:
      assert(0);
  }
```

Figure 4. Excerpt from a SHA implementation. The assertion depends on the result of a division

the property (at least one of the four predicates holds). The overall run-time (including preparation, one refinement iteration, and the SMV run-time) is 24 seconds on a 2 GHZ machine, most of which is spent within the SAT solver. The predicate abstraction tools described in the related work generate an abstraction that lacks at least the last property, i.e., that the result of the division is one of 0 to 3.

5.2. ASN1 Data Structures in OpenSSL

OpenSSL comes with an implementation of ASN1 data structures for managing certificates. Using an if-then-else construct, individual bits of a variable `j` are tested. The variable has type `signed int`. Previously, the integer is assigned the value of an unsigned char array member. The array member is known to be non-zero. The code assumes that therefore one of the first eight bits must be set (figure 5).

Within one refinement iteration, the following predicates are obtained: one predicate that holds if the array member is non-zero and one predicate for each of the branching guards. Using these predicates, our tool generates an abstract transition relation which enforces that exactly one of these predicates is true, which allows the model checker to show that the assertion is not reachable.

```
int ret,j,bits,len;
. . .

  j=a->data[len-1];
  if      (j & 0x01) bits=0;
  else if (j & 0x02) bits=1;
  else if (j & 0x04) bits=2;
  else if (j & 0x08) bits=3;
  else if (j & 0x10) bits=4;
  else if (j & 0x20) bits=5;
  else if (j & 0x40) bits=6;
  else if (j & 0x80) bits=7;
  else { bits=0; assert(0); } /* should not happen */
```

Figure 5. Excerpt from an implementation of ASN1 data structures from OpenSSL. Proving the assertion requires a bit-vector decision procedure. The assertion is not part of the original code.

5.3. MD2 Message-Digest Algorithm

Similar to the SHA algorithm, the MD2 message-digest algorithms computes a hash of a given message. RFC 1319 gives a reference implementation in ANSI-C. A part of it is shown in figure 6. The algorithm makes extensive use of a permutation that is given as an array. In the first part, the result of the previous iteration is used as an array index for the next iteration. The second part uses the bit-wise xor of the result of the previous iteration and a part of the message as an array index.

We verify that these lookups do not violate the bounds of the `PI_SUBST` array. As the variable `t` is of an unsigned integer type, only the upper array bound can be violated, i.e., the predicate `t<256` must hold in the first part, and the predicate `(block[i]^t)<256` must hold in the second part of the algorithm. For each of the four code locations `t` is modified in, the SAT solver easily discovers that these predicates indeed are true in the next state.

5.4. Pointer Arithmetic in JPEG Decoder

For efficiency reasons, many programs use pointer arithmetic instead of array index expressions within loops. As an example, consider the code in figure 7. It performs a discrete cosine transformation using a loop that iterates through an array of 64 elements. Each loop iteration processes one row, which corresponds to `DCTSIZE=8` array elements. Thus, iteration number `ctr` accesses the elements `data[8*(7-ctr)]` to `data[8*(7-ctr)+7]`. In order to avoid this computation for each array access, the code uses a pointer that points to `data[8*(7-ctr)]`. This pointer is then used to access the individual elements.

```
static unsigned char PI_SUBST[256] =
   ...
;

static void MD2Transform (state, checksum, block)
unsigned char state[16];
unsigned char checksum[16];
unsigned char block[16];
{
  unsigned int i, j, t;
  unsigned char x[48];

  /* Form encryption block from state, block, state ^ block.
   */
  MD2_memcpy ((POINTER)x, (POINTER)state, 16);
  MD2_memcpy ((POINTER)x+16, (POINTER)block, 16);
  for (i = 0; i < 16; i++)
    x[i+32] = state[i] ^ block[i];

  /* Encrypt block (18 rounds).
   */
  t = 0;
  for (i = 0; i < 18; i++) {
    for (j = 0; j < 48; j++)
      t = x[j] ^= PI_SUBST[t];   /* t must be <= 255 */
    t = (t + i) & 0xff;
  }

  /* Save new state */
  MD2_memcpy ((POINTER)state, (POINTER)x, 16);

  /* Update checksum.
   */
  t = checksum[15];
  for (i = 0; i < 16; i++)
    t = checksum[i] ^= PI_SUBST[block[i] ^ t]; /* t must be <= 255 */

  /* Zeroize sensitive information.
   */
  MD2_memset ((POINTER)x, 0, sizeof (x));
}
```

Figure 6. Excerpt from an the reference implementation of the MD2 algorithm.

In order to prove that the pointer access happens within the array bounds, we use the predicates `dataptr==&data[8*(7-ctr)]`, `ctr>=0`, and `ctr<DCTSIZE`.

6. Conclusion

This paper presented a new method to compute the predicate abstraction of an ANSI–C program. This new method replaces the use of theorem provers with the use of a SAT solver. We suggest that SAT-based predicate abstraction

```
jpeg_fdct_ifast (DCTELEM * data)
{
  ...
  DCTELEM *dataptr;
  int ctr;
  ...

  /* Pass 1: process rows. */

  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[0] + dataptr[7];
    ...

    dataptr += DCTSIZE; /* advance pointer to next row */
  }
  ...
```

Figure 7. Excerpt from an JPEG decoder.

outperforms the approaches that use theorem provers since enumeration on a single SAT instance can substitute for a potentially exponential number of theorem prover calls. The advantages are particularly pronounced when the number of abstract transitions is significantly smaller than the number of possibilities that need to be checked. Furthermore, since modern SAT solvers allow for the evaluation of a large number of possible assignments to the abstract program variables, the application of a SAT engine results in a more precise transition relation of the abstract program compared to the abstraction produced by using theorem provers. This results in eliminating some unrealistic behaviors of the abstract program that otherwise would be introduced during the over-approximations of the abstract transition relation computed using a theorem prover.

Model checking a more precise abstract program, therefore, exhibits a smaller number of redundant spurious counterexamples. As a result, a smaller number of the CEGAR loop iterations is required until the verification property is confirmed or refuted. The latter fact is of high value to practical software verification since the validation of counterexamples and predicate refinement (Steps 3 and 4 of the CEGAR loop) are computationally expensive. Our approach, therefore, simplifies (if not enables) the application of model checking to the verification of large-scale programs by eliminating analysis and refinement of redundant counterexamples.

Another contribution of the SAT-based abstraction technique is that most ANSI-C constructs can be handled during the program abstraction. This differs from other model checking approaches that operate only on a small subset of the C language. Our approach enables model checking of realistic programs by supporting the more complex features of C, such as multiplication/division, pointers, bit-wise operations, type conversion and shift operators.

A notable advantage of the SAT-based abstraction technique is that it can be reused within the CEGAR loop without any changes to the error trace simulation and predicate refinement used in the loop.

In the future, we plan to use the ideas presented here in other parts of the CEGAR loop. That is, we would be interested to use the SAT enumeration engine to conduct predicate discovery for the refinement of the abstracted program. We also plan to implement the abstraction of floating point arithmetic, as well as to extend the technique to the verification of concurrent programs.

References

1. Ball, T., R. Majumdar, T. Millstein, and S. Rajamani: 2001, 'Automatic Predicate Abstraction of C Programs'. In: *SIGPLAN Conference on Programming Language Design and Implementation*. pp. 203–213.
2. Ball, T. and S. Rajamani: 2000, 'Boolean Programs: A Model and Process for Software Analysis'. Technical Report 2000-14, Microsoft Research.
3. Ball, T. and S. K. Rajamani: 2001, 'Automatically Validating Temporal Safety Properties of Interfaces'. In: *The 8th International SPIN Workshop on Model Checking of Software, LNCS vol. 2057*. pp. 103–122.
4. Bensalem, S., Y. Lakhnech, and S. Owre: 1998, 'Computing Abstractions of Infinite State Systems Compositionally and Automatically'. In: A. J. Hu and M. Y. Vardi (eds.): *Computer-Aided Verification, CAV '98*, Vol. 1427. Vancouver, Canada, pp. 319–331.
5. BOOP. http://boop.sourceforge.net/.
6. Brat, G., K. Havelund, S. Park, and W. Visser: 2000, 'Java PathFinder - A Second Generation of a Java Model Checker'. In: *Workshop on Advances in Verification, Chicago, Illinois*. pp. 130–135.
7. Chaki, S., E. Clarke, A. Groce, S. Jha, and H. Veith: 2003a, 'Modular Verification of Software Components in C'. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. pp. 385–395.
8. Chaki, S., E. Clarke, A. Groce, and O. Strichman: 2003b, 'Predicate Abstraction with Minimum Predicates'. In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*.
9. Clarke, E., O. Grumberg, S. Jha, Y. Lu, and V. H.: 2000, 'Counterexample-Guided Abstraction Refinement'. In: *Computer Aided Verification*. pp. 154–169.
10. Clarke, E., O. Grumberg, and D. Long: 1992, 'Model Checking and Abstraction'. In: *Principle of Programming Languages*.
11. Clarke, E., O. Grumberg, and D. Peled: 1999, *Model Checking*. MIT Press.
12. Clarke, E., M. Talupur, and D. Wang: 2003, 'SAT based Predicate Abstraction for Hardware Verification'. In: *Sixth International Conference on Theory and Applications of Satisfiability Testing*.
13. Colon, M. and T. Uribe: 1998, 'Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures'. In: *Computer Aided Verification*. pp. 293–304.
14. Cousot, P. and R. Cousot: 1977, 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Principles of Programming Languages, POPL '77*. pp. 238–252.

15. Currie, D. W., A. J. Hu, S. Rajan, and M. Fujita: 2000, ‘Automatic Formal Verification of DSP Software’. In: *37th ACM/IEEE Design Automation Conference*. pp. 130–135.
16. Dams, D., R. Gerth, and O. Grumberg: 1997, ‘Abstract Interpretation of Reactive Systems’. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **19**(2).
17. Das, S. and D. Dill: 2001, ‘Successive approximation of abstract transition relations’. In: *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*.
18. Detlefs, D., G. Nelson, and J. B. Saxe: 2003, ‘Simplify: A theorem prover for program checking’. Technical Report HPL-2003-148, HP Labs.
19. Digital Signature Standard: 1995, ‘Secure Hash Standard (FIPS 180-1)’. National Institute of Standards and Technology.
20. Graf, S. and H. Saidi: 1997, ‘Construction of Abstract State Graphs with PVS’. In: O. Grumberg (ed.): *Proc. 9th INternational Conference on Computer Aided Verification (CAV’97)*, Vol. 1254. pp. 72–83.
21. Gries, D. and G. Levin: 1980, ‘Assignment and procedure call proof rules’. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **2**(4), 564–79.
22. Gupta, A., Z. Yang, P. Ashar, and A. Gupta: 2000, ‘SAT-Based Image Computation with Application in Reachability Analysis’. In: *Formal Methods in Computer-Aided Design (FMCAD)*. pp. 354–372.
23. Henzinger, T. A., R. Jhala, R. Majumdar, and G. Sutre: 2002, ‘Lazy abstraction’. In: *Symposium on Principles of Programming Languages*. pp. 58–70.
24. Kesten, Y. and A. Pnueli: 2000, ‘Control and Data Abstraction: cornerstones of the practical formal verification’. *Software Tools and Technology Transfer* **2(4)**, 328–342.
25. Kroening, D., E. Clarke, and K. Yorav: 2003a, ‘Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking’. In: *40th Desgin Automation Conference*. pp. 368–371.
26. Kroening, D., E. Clarke, and K. Yorav: 2003b, ‘Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking’. Technical Report CMU-CS-03-126, Carnegie Mellon University.
27. Kurshan, R.: 1994, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press.
28. Lahiri, S. K., R. E. Bryant, and B. Cook: 2003, ‘A Symbolic Approach to Predicate Abstraction’. In: W. A. Hunt and F. Somenzi (eds.): *Computer-Aided Verification (CAV)*. pp. 141–153.
29. Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem: 1995, ‘Property Preserving Abstractions for the Verification of Concurrent Systems’. *Formal Methods in System Design* **6**, 11–45.
30. McMillan, K.: 2002, ‘Applying SAT Methods in Unbounded Symbolic Model Checking’. In: *14th Conference on Computer Aided Verification*. pp. 250–264.
31. Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik: 2001, ‘Chaff: Engineering an Efficient SAT Solver’. In: *Proceedings of the 38th Design Automation Conference (DAC’01)*. pp. 530–535.
32. Plaisted, D.: 2000, ‘Method for design verification of hardware and non-hardware systems’. United States Patent, 6,131,078.
33. Plaisted, D., A. Biere, and Y. Zhu: 2003, ‘A Satisfiability Tester for Quantified Boolean Formulae’. *Journal of Discrete Applied Mathematics (DAM)*. In press, available online.
34. SMV. http://www-2.cs.cmu.edu/˜modelcheck/smv.html.