Theoretical Computer Science 404 (2008) 256–274

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Efficient SAT-based bounded model checking for software verification☆

Franjo Ivančić[a,*], Zijiang Yang[b], Malay K. Ganai[a], Aarti Gupta[a], Pranav Ashar[a]

[a] NEC Laboratories America, 4 Independence Way, Ste. 200, Princeton, NJ 08540, United States

[b] Department of Computer Science, Western Michigan University, 1903 W. Michigan Ave., Kalamazoo, MI 49008, United States

ARTICLE INFO

*Keywords:*
Software verification
Bounded model checking
SAT-based model checking

ABSTRACT

This paper discusses our methodology for formal analysis and automatic verification of software programs. It is applicable to a large subset of the C programming language that includes pointer arithmetic and bounded recursion. We consider reachability properties, in particular whether certain assertions or basic blocks are reachable in the source code, or whether certain standard property violations can occur. We perform this analysis via a translation to a Boolean circuit representation based on modeling basic blocks. The program is then analyzed by a back-end SAT-based bounded model checker, where each unrolling is mapped to one step in a block-wise execution of the program.

The main contributions of this paper are as follows: (1) Use of basic block-based unrollings with SAT-based bounded model checking of software programs. This allows us to take advantage of SAT-based learning inherent to the best performing bounded model checkers. (2) Various heuristics customized for models automatically generated from software, allowing a more efficient SAT-based analysis. (3) A prototype tool called F-Soft has been implemented using our methodology.

We present experimental results based on multiple case studies including a C-based implementation of a network protocol, and compare the performance gains using the proposed heuristics.



## 1. Introduction

Model checking is an automatic technique for the verification of concurrent systems. It has several advantages over simulation, testing, and deductive reasoning, and has been used successfully in practice to verify complex sequential circuit designs and communication protocols [15]. In particular, model checking is automatic, and, if the design contains an error, model checking produces a counterexample, i.e., a witness of the offending behavior of the system that can be used for effective debugging of the system. The procedure normally uses an exhaustive search of the state-space of the considered system to determine whether a specification is true or false. A brief overview of model checking techniques is provided in Section 2.

While model checking of hardware designs and protocols has been extensively studied, its application to software verification had been limited to the use of specialized modeling languages to capture program semantics. The capability of directly model checking source code programs written in popular programming languages, such as C/C++ and Java, is relatively new [67]. The general approach is to extract suitable verification models from the given source code

☆ A preliminary version of this paper appeared in the 1st Intern. Symposium for Leveraging Applications of Formal Methods (ISoLA), in November 2004 (see [F. Ivančić, Z. Yang, M. Ganai, A. Gupta, P. Ashar, Efficient SAT-based bounded model checking for software verification, in: Symposium on Leveraging Formal Methods in Applications, ISoLA, 2004]).

* Corresponding author.
*E-mail address:* ivancic@nec-labs.com (F. Ivančić).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.03.013

programs, on which back-end model checking techniques are applied to perform verification. Given the popularity of these languages, and the increasing costs of software development, verifying programs directly written in these languages is very attractive in principle. However, there are many challenging issues — handling of integers/floating point data variables, pointers, recursion and function/procedure calls, concurrency, object-oriented features such as classes, dynamic objects, and polymorphism. Different choices can be made in modeling these features in terms of accuracy, resulting in various trade-offs. Some of these are described for C programs in Section 3.

The overall focus is usually on reducing the size of the resulting verification models, by use of appropriate abstractions, in order to manage verification complexity. The two important measures to keep in mind are: *soundness*, i.e. any property proved true is indeed true (no false positives); and *completeness*, i.e. any property that is true can be proved true (no false negatives). Typically, modeling and abstraction techniques may sacrifice completeness in practice (even if guaranteed in principle) due to loss of precision in the abstract models. Furthermore, much useful high-level information may be lost during the translation from programs to a verification model. Therefore, several software model checkers make a special effort to exploit high-level information such as control flow and procedure/function boundaries, both during translation to and during analysis of the verification models. Such use of high-level information in back-end model checkers is described in Section 5.

In terms of abstraction techniques, *predicate abstraction* has emerged to be a popular technique for extracting verification models from software [4,20,22,30]. Predicate abstraction is used to abstract out data, by keeping track of predicates which capture relationships between data variables in the program. In the abstract model, each predicate is represented by a Boolean variable, while the original data variables are eliminated. In this way, predicate abstraction allows translation of a given concrete model to an abstract model, which simulates the concrete model but is usually much smaller. Due to conservative abstraction, the abstract model has many more behaviors than the concrete model. Therefore, correctness of a property on the abstract model guarantees correctness on the original concrete model. However, a property shown to be false on the abstract model needs further investigation. In particular, an abstract model may contain the so-called *spurious* counterexamples, that do not correspond to any feasible counterexample in the concrete model. Such spurious counterexamples can be eliminated by generating a *refinement* of the abstraction. This process of abstraction and refinement can be iterated until the property is either proved correct on the abstract model (thereby guaranteeing that it is also correct on the concrete model) or disproved (by demonstrating existence of a real counterexample on the concrete model). Such techniques are similar to *counterexample-guided abstraction refinement* [14,46] demonstrated for hardware designs.

### *1.1.* F-Soft

In this paper, we describe our overall methodology for the analysis of software programs. We have developed a prototype software model checking tool called F-Soft [40], which has been used primarily for verification of sequential C programs. It considers reachability properties for verification, in particular whether certain labeled statements are reachable in the program. It also includes automatic checkers for a set of standard programming bugs such as array bound violations, NULL pointer dereferences, use of uninitialized variables, memory leaks, improper use of C string library functions, lock/unlock violations, division by zero, etc. These checkers are implemented by automatically adding property monitors to the given source code programs. Verification is performed via a translation of the given C program to a finite state circuit model, derived automatically by considering the control and data flow of the program (under the assumptions of bounded data and bounded recursion). Optionally, predicate abstraction is supported by a fully automated abstraction refinement framework. The back-end model checking is performed by a tool called VeriSol (based on DiVer) [26], which includes several state-of-the-art symbolic model checking techniques. F-Soft also contains a multitude of static program analysis engines [60,61] to find easy proofs and simplify the model of the program as much as possible before invoking the model checker on the simplified model and the remaining properties.

A salient feature of our approach for software verification is the central role played by a basic block (sequence of non-branching statements). From the modeling of software, the abstraction of the source code, to the verification of the generated software model using an unrolling based on basic blocks — the basic block modeling approach is integral to the proposed method.

A key contribution of this paper is to use the program counter variable to track progress of the allowed executions of the code during SAT-based BMC. Effectively, an unrolling during BMC is understood to be one step in a block-wise execution of the program. In this context each atomic step of the BMC consists of a basic block rather than individual statements. This is an efficient way of performing BMC, since for each basic block there are only a limited number of possible successors. Given a single initial basic block, there are thus only a limited number of possible next blocks reachable in new unrollings. Thus, although each unrolling introduces all basic blocks into the satisfiability problem, many basic blocks in the new unrolling can be declared unreachable by merely considering the control flow of the software program. This intuition can be used to prune the search space considerably.

### *1.2. Related work*

The most closely related work to ours is the CBMC tool [16,45], which also translates a C program into a Boolean representation to be analyzed by a back-end SAT-based BMC. However, there are many differences in our approaches. One

major difference is that we generate a Boolean model of the software that can be analyzed by both SAT-based bounded and unbounded model checking, and model checking using BDDs. Another major difference is the block-based approach, rather than a statement-based approach. Other related work describes usage of basic blocks for predicate abstraction of C programs [17]. Additionally, CBMC requires a full inlining of functions with no recursion and unwinding of the source code through a user-supplied constant. This allows the SAT-solver to consider only loop-free paths which is often fast for small pieces of code with no or few loops. However, this method of inlining and unwinding loops is clearly not scalable to larger pieces of code or to code with reactive behavior. Our translation method does not require multiple inlining or unwinding of loops, and can handle bounded recursion.

We also differentiate our approach through the use of pre-processing analyses such as *program slicing, range analysis* and *invariant analysis using numeric domains*. A slice of a program with respect to a set of program elements is a projection of the program that includes only program elements that might affect the property [13,18,44,66]. Furthermore, we use range analysis techniques [58,71] to limit the number of bits needed to represent various variables in the Boolean model compared to using a full bit-width encoding as implemented in CBMC. We also do not need to rewrite the source code to single assignment form, thus keeping only one copy of each variable. Finally, we use inter-procedural invariant computations using the *octagon numeric domain* [54] to simplify the program before applying model checking [60,61].

We propose new heuristics to exploit many common features of Boolean models generated by our software modeling approach. We discuss three main heuristics which prune the SAT search space considerably, and the variations we have used for combining them. Our tool also allows abstraction of the software using localized counterexample-guided predicate abstraction [41] based on predicate localization [35] and symbolic predicate abstraction techniques [17,47]. In addition, we use a priori statically computed invariants to discover some relationships cheaply, thereby improving the overall performance of the predicate abstraction refinement loop [42]. In this paper, we focus the discussion on our software modeling framework and our improved SAT-solver decision heuristics.

### 1.3. Overview

We start by providing a brief background on model checking in Section 2. In Section 3 we discuss our block-based software modeling approach. Section 4 shows how to construct the Boolean circuit representation from a model. Section 5 discusses our analysis using SAT-based BMC and proposes customized decision heuristics that improve the efficiency of the analysis. In Section 6 we present the prototype software verification tool F-Soft [40]. Section 7 introduces one of the analyzed network protocol case studies, while Section 8 presents experimental results for various case studies using F-Soft in detail. Section 9 concludes this paper with some remarks and pointers to future work.

## 2. Background: Model checking

This section provides a brief overview and terminology for model checking — more details can be found in the related book [15]. Model checking is a popular technique for checking correctness properties, in which the design to be verified is represented as a finite state transition system, and the property is specified as a temporal logic formula. Temporal logics are very useful for specifying dynamic behavior over time. Different variants of temporal logics have become popular, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), depending on whether a linear or a branching view of time is considered, respectively. In this paper, we focus mainly on simple safety properties, denoted in CTL as **AG***p*. This formula specifies that on *all (A)* paths of a system, *globally (G)* in each state of the path, the property $p$ holds. Such properties can be verified by an exhaustive traversal of the state-space to check that $p$ holds in every reachable state, i.e. it is an invariant. Alternately, safety properties can also be checked by searching for a counterexample, which shows reachability of an unsafe state (where $p$ is false).

Model checking can be applied directly for verification of finite state systems, such as sequential circuits and protocol controllers. In addition, by use of suitable abstractions, finite state models can also be extracted from infinite state systems, for subsequent verification using model checking. These applications include real-time system verification [1], parameterized system verification [9], and software program verification [18,67]. Furthermore, model checking techniques have also been extended to pushdown systems [5,8], i.e. systems with a finite control but with an unbounded stack. Such systems allow a direct modeling of recursion inherent in software programs. In this paper, we will focus on techniques for extracting finite state models from C programs, as described in detail in Section 3. These ideas also apply to extraction of pushdown models, such as Boolean programs [5].

Explicit state model checkers, such as SPIN [37], use an explicit representation of states and transitions in the system, and enumerate all reachable states explicitly. They utilize many additional techniques such as state hashing for compaction of state representations, and partial order methods to avoid exploring all interleavings of concurrent processes. The scalability issue in explicit state enumeration makes these checkers unsuitable for hardware designs, although they have found practical success in verification of controllers and software.

In contrast, symbolic model checkers, such as SMV [49], avoid an explicit enumeration of the state-space by using symbolic representations of sets of states and transitions. They typically use *Binary Decision Diagrams* (BDDs) [10], which provide a canonical symbolic representation of Boolean formulas and efficient graph-based algorithms for symbolic

manipulation. For hardware designs, where these symbolic representations effectively capture the regularity in the state-space, symbolic model checking has significantly extended the ability to handle large state-spaces.

Despite the considerable benefits of symbolic model checking using BDDs, the basic *verification* approach of exhaustive analysis does not scale well in practice. An alternative is the use of *falsification* approaches, such as *bounded model checking* (BMC) [6], which focus primarily on the search for finding bugs. In BMC, the problem of searching for a counterexample of length $k$ is translated to a Boolean formula (by unrolling the transition relation of the design $k$ times), such that the formula is *satisfiable* if and only if there exists a counterexample of length $k$. In practice, $k$ can be increased incrementally to find a shortest counterexample if one exists. Additional reasoning, in the form of completeness thresholds [6] or proofs by induction [31,63], can be combined with BMC to ensure completeness when desired.

The Boolean satisfiability (SAT) check in the BMC approach is typically performed by a back-end *SAT-solver*. Most modern SAT-solvers use a DPLL-style-search-based decision procedure, with distinct methods and heuristics for making decisions (choosing a variable and value to explore), for Boolean constraint propagation (making implications on other variables), and for performing conflict analysis and backtracking in case a conflict is found. Due to many recent advances in SAT-solvers [48,55], verification techniques based on SAT have become very popular (see a recent survey [56] for useful pointers). In particular, SAT-based BMC is often successful in finding bugs in much larger hardware designs than BDD-based approaches, and has also been used successfully for verifying C programs [16,40]. A related important development has been the use of *resolution-based proof-analysis* techniques [28,72] for SAT-solvers. These techniques were developed in order to independently check the unsatisfiability result of a SAT-solver. In addition, these techniques can also identify a set of clauses from the original problem, called the *unsatisfiable core*, that are sufficient for implying unsatisfiability. The unsatisfiable core has been used very effectively for proof-based abstraction [32,52], refinement [11], and for interpolant-based verification [35,51]. These methods allow SAT-based BMC to be combined effectively with other techniques to provide complete verification methods. There has also been growing interest in the use of SAT for unbounded model checking [25, 33,50]. However, these techniques are not as robust as SAT-based BMC techniques.

## 3. Software modeling for C programs

Symbolic model checkers (both SAT- and BDD-based) work on a symbolic transition relation of a finite state system, typically represented in terms of a vector of binary-valued *latches* and a Boolean next state function (or relation) for each latch. For reachability properties, unsafe states can be specified quite simply as a predicate on the latches. In this section, we describe an approach (implemented in the F-Soft tool) for translating a given C program into a finite state model whose traces represent C program traces, and to represent this model symbolically using binary-valued latches and their transition relations. In other words, all high-level C constructs (arrays, pointers, dynamic memory, control flow) require faithful modeling, ultimately in terms of binary-valued latches and Boolean functions. Note that high-level synthesis systems, i.e., systems that synthesize RTL hardware descriptions from high-level C specifications also face this task, although they need to handle only a subset of C sufficient for describing hardware [62,65].

### 3.1. Labeled transition graphs

We begin with full-fledged C and apply a series of source-to-source transformations into smaller subsets of C, until program state is represented as a collection of simple scalar variables and each program step is represented as a set of parallel assignments to these variables. Formally, the transformations produce a labeled transition graph (LTG) of the program. A LTG $G$ is a tuple $\langle B, b_s, E, X, I, C, A, \delta, \theta\rangle$, where

- $B = \{b_1, \ldots, b_n\}$ is a finite non-empty set of basic blocks.
- $b_s \in B$ is the initial basic block.
- $E \subseteq B \times B$ is the set of edges. If $(b_i, b_j) \in E$, we say $b_i$ is a predecessor of $b_j$, and $b_j$ is a successor of $b_i$.
- $X$ is a finite set of program variables.
- $I$ contains initialization constraints on variables in $X$.
- $C$ is a finite set of conditions that determines control flow of a C program. Typical sources include the conditional expressions in `if-then-else, while, for` statements.
- $A$ is a finite set of assignments.
- $\delta : B \to 2^A$ is a labeling function that labels each basic block with a set of parallel assignments. We denote a type-consistent valuation of all variables in $X$ by $\boldsymbol{x}$, and the set of all type-consistent valuations by $\mathcal{X}$. Let the set of allowed C-expressions be denoted by $\Sigma$. Then, the parallel assignments in each basic block can be written as $Y \leftarrow e_1, \ldots, e_n$, where $Y = (y_1, \ldots, y_n)$, $\{y_1, \ldots, y_n\} \subseteq X$ and $\{e_1, \ldots, e_n\} \subseteq \Sigma$.
- $\theta : E \to C$ is a labeling function that labels each edge with a condition. For the Boolean circuit model we require that at most one outgoing guard is satisfied at any point. Given a basic block $b_i$, let $B_i \subset B$ be the set of all successors of $b_i$. Then $\forall b, b' \in B_i : b \neq b' \Longrightarrow \theta(b_i, b) \wedge \theta(b_i, b') \equiv 0$.

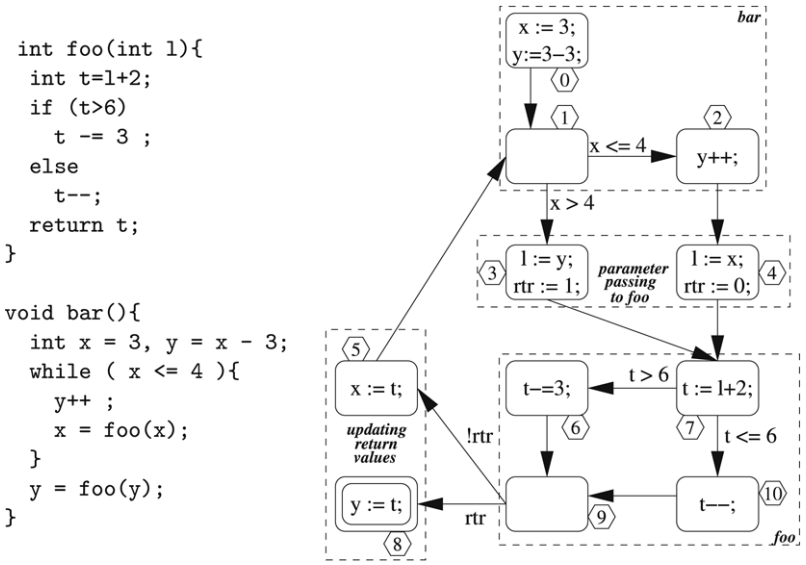


**Fig. 1.** Labeled transition graph.

As a running example, Fig. 1 shows an LTG $G$ obtained from the C program on the left side. Each rectangle is a basic block with an associated unique number showing its index. i.e., $B = \{b_0, \ldots, b_{10}\}$. The assignments inside each basic block are obtained by applying labeling function $\delta$. Note that a basic block can be empty (e.g. $\delta(b_1) \equiv \emptyset$). The edges are labeled by conditional expressions; e.g. $\theta(b_1, b_2) \equiv x \leq 4$. When an edge is not labeled by any condition, the default condition is true. The example pictorially shows how non-recursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`.

### 3.2. Modeling of C program memory

One of the biggest difficulties in modeling C programs lies in modeling indirect memory accesses via pointers, such as $x = *(p+i)$ or $*(q+j) = y$. This includes array accesses, since $A[e]$ is equivalent to $*(A+e)$. We replace all indirect accesses in the C program with expressions involving only direct variable accesses by introducing appropriate conditional expressions on auxiliary variables as described in detail below.

**Modeling the heap and stack.** The C language specification does not bound heap or stack size, but our focus is on generating a bounded model only. Therefore, we model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into this array. We also add a bounded depth stack as another global array, in order to handle bounded recursion, if required, along with code to save and restore local state for recursive functions only. Note that in practice we generally analyze embedded software where we generally do not observe extensive use of recursion.

**Modeling pointers.** We build an internal memory representation of the program by assigning to each variable a unique number representing its memory address. Variables that are adjacent in C program memory (for example, adjacent elements of one array) are given consecutive memory addresses in our model; this facilitates the modeling of pointer arithmetic. Pointers are modeled as integers: pointer variable $p$ points to simple variable $x$ by storing the integer memory address assigned to $x$.

We perform a points-to analysis [36] to determine, for each indirect memory access, the set of variables that may be accessed (called the *points-to set*). If we determine that pointer $p$ can point to variables $a, b, \ldots, z$ at a given program location, we can rewrite a pointer read $*(p+i)$ as a conditional expression of the form $((p+i) == \&a ? a : ((p+i) == \&b ? b : \ldots))$ where $\&a, \&b, \ldots$ are the numeric memory addresses we assigned to the variables $a, b, \ldots$ respectively.

**Inferred variables.** Additional variables are introduced when pointer variables are declared. The declaration `int **p;` creates three variables $v_p, v'_p, v''_p$, where $v_p$ stands for `p`, $v'_p$ for `*p`, and $v''_p$ for `**p`. Similarly, the declaration `int *a, *b;` creates four variables $v_a, v'_a, v_b, v'_b$, while a dereference in the C code, such as `&a`, also leads to an additional variable — in this case the variable $'v_a$.

In our memory modeling framework we may thus have several copies of the same value although they all represent the same location. Although this modeling approach may increase the number of variables when compared to the usual heap model, it can lead to analysis savings. This is due to the fact that the number of live variables can often be substantially reduced when a variable is read through a pointer dereference, because a pointer variable can point only to one location at a time, even though its points-to set may be large. Consider the example of a pointer `p` that could point to any element of a set of variables of size $N$. The statement `*p` usually requires all $N$ elements to be live at this point, since any single element

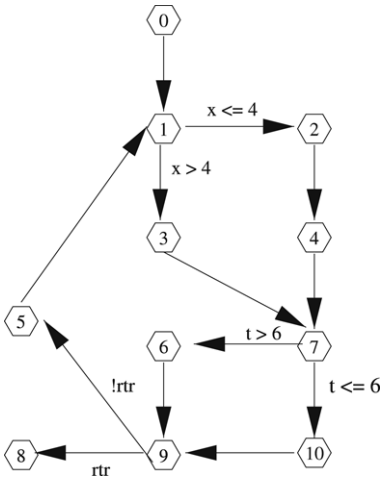


**Fig. 2.** Control logic subgraph .

may be pointed to by p. However, by adding a dedicated variable that always keeps the value of *p updated, we can reduce the number of live variables important at this point to this single dedicated variable. This observation has been reported in hardware synthesis frameworks [62], and we believe we are the first to exploit it for verification purposes. Analysis methods, such as SAT-solvers, that can exploit the effect of fewer live variables while being relatively unaffected by the increase in problem size can thus scale better in practice.

**Inferred assignments.** In our memory modeling framework, additional assignments have to be inferred due to aliasing and newly introduced variables. We distinguish between two types of additional assignments, namely assignments due to aliasing and implied assignments. Assignments based on aliasing are due to the fact that multiple copies of the same location may exist. Implied assignments are due to the fact that we explicitly model the values of the pointed to locations for pointer variables.

As an example, consider the assignment p=&a. This directly corresponds to the assignment $v_p = {}'v_a$. However, since p=&a implies *p=a and **p=*a, two new assignments $v'_p = v_a$ and $v''_p = v'_a$ are also inferred from the original assignment in the source code. We call such assignments implied assignments.

An assignment a=&x gives rise not only to the assignments $v_a = {}'v_x$ and $v'_a = v_x$, but also to conditional assignments due to aliasing. Since p may equal &a, it is possible that *p and **p are assigned new values when a is assigned. This results in the conditional assignments $v'_p = (p == {}'v_a)?'v_x : v'_p$ (which stands for *p=(p==&a)?&x:*p) and $v''_p = (p == {}'v_a)?v_x : v''_p$ (which stands for **p=(p==&a)?x:**p). These kind of assignments are called assignments due to aliasing.

## 4. Boolean representation

In this section we describe how to generate a Boolean transition relation from an LTG $G$. We define a *state* of a program to be a tuple $(b, \boldsymbol{x})$, consisting of a location $b \in B$ representing the basic block, and a type-consistent valuation of data variables $\boldsymbol{x} \in \mathcal{X}$, where out-of-scope variables at $b$ are assigned the undefined value $\perp$. We consider the initial state of the program to be an initial location $b_s$, where each variable in $X$ can take any value that is type-consistent with its specification. For checking reachability in programs, we define a set of blocks Bad $\subseteq B$ to be unsafe, and model checking is used to prove or disprove that these basic blocks can be reached. Formally, we define a path of length $k$ in the state space to be a sequence of $k$ states $(b_0, \boldsymbol{x_0}), \ldots, (b_{k-1}, \boldsymbol{x_{k-1}})$ such that $b_0 \equiv b_s$ is the initial basic block and $\forall 0 \leq i < k-1 : (b_i, \boldsymbol{x_i}) \rightarrow (b_{i+1}, \boldsymbol{x_{i+1}})$, where $\rightarrow$ denotes a transition between the states. A counterexample of length $k$ is a path that ends in an unsafe location, that is $b_{k-1} \in$ Bad.

In order to construct a Boolean transition relation of a LTG $G$, we divide $G$ into two subgraphs $G_C$ and $G_D$, where $G_C$ captures the control flow and $G_D$ captures the data logic.

### 4.1. Control logic

Given an LTG $G = \langle B, b_s, E, X, I, C, A, \delta, \theta \rangle$, we define a control logic subgraph $G_C = \langle B, b_s, E, X, C, \theta \rangle$. $G_C$ discards the labeling function $\delta$ and assignment set $A$ because it does not care how variables are updated. However, it keeps all the labeled edges between the basic blocks in order to capture the control flow information in a C program. A program counter ($pc$) variable is introduced to monitor progress in $G_C$. For example, Fig. 2 shows the control logic subgraph for the program shown in Fig. 1.

One contribution of this paper is to use the program counter to track progress of allowed executions of the code during BMC. The model checking analysis is performed by understanding an unrolling to be one step in a block-wise execution of

**Table 1**
Truth table for control logic

| $c_1$ | $c_2$ | $\cdots$ | $c_n$ | $c_1'$ | $c_2'$ | $\cdots$ | $c_n'$ | Guard |
|---|---|---|---|---|---|---|---|---|
| $v_1^1$ | $v_2^1$ | $\cdots$ | $v_n^1$ | $v_1^{1'}$ | $v_2^{1'}$ | $\cdots$ | $v_n^{1'}$ | $k^1$ |
| $v_1^2$ | $v_2^2$ | $\cdots$ | $v_n^2$ | $v_1^{2'}$ | $v_2^{2'}$ | $\cdots$ | $v_n^{2'}$ | $k^2$ |
| | $\cdots$ | $\cdots$ | | | $\cdots$ | $\cdots$ | | $\cdots$ |
| $v_1^m$ | $v_2^m$ | $\cdots$ | $v_n^m$ | $v_1^{m'}$ | $v_2^{m'}$ | $\cdots$ | $v_n^{m'}$ | $k^m$ |

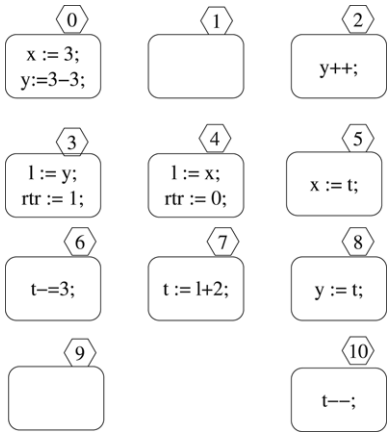


**Fig. 3.** Data logic subgraph.

the program, where each atomic step consists of a basic block rather than an individual statement. Similar approaches have been explored in a non-BMC setting to software model checking (such as with tools like VeriSoft [27], Java PathFinder [34, 67] and Bogor [57]), which have also extended it to deal with concurrency by using partial-order reduction. However, by incorporating this idea into a SAT-based BMC we are able to take advantage of recent progress in SAT-solvers, while also improving its efficiency for software verification by customizing the SAT-solver heuristics.

Next we show how to extract the control logic that defines the transition relation for the $pc$ with the edges and the conditions guarding the transitions between the blocks in $G_C$. If $N$ denotes the number of basic blocks in a $G_C$, we can use $2\lceil \log N \rceil$ bits to express the program counter. Let $c_1, c_2, \cdots c_n$ denote the current state program counter bits, and $c_1', c_2', \cdots c_n'$ denote the next state program counter bits where $n = \lceil \log N \rceil$.

An edge $E_{ij} = (b_i, b_j)$ is enabled if and only if $pc = i \wedge \theta(b_i, b_j) = true$. Once $E_{ij}$ is enabled, the next value for $pc$ is $j$. i.e., $pc' = j$. $G_C$ is encoded in Table 1, where the first $n$ columns show the bit values of current state $pc$ variables, the next $n$ columns show the bit values of next state $pc$ variables, and the last column shows the bit value of the guarding condition. The $j$th table line represents an edge $(v_1^j v_2^j \cdots v_n^j \rightarrow v_1^{j'} v_2^{j'} \cdots v_n^{j'})$ in the control flow graph with guard $k^j$, where $v_i^j \in \{0, 1\}$ is an assignment to $c_i$ and $v_i^{j'} \in \{0, 1\}$ is an assignment to $c_i'$. Based on the truth table, we build the next state logic for each program counter bit as:

$$c_i' = \bigvee_{j:v_i^{j'}=1} \left( k^j \wedge \bigwedge_{p:v_p^j=1} c_p \wedge \bigwedge_{p:v_p^j=0} \neg c_p \right).$$

Finally, the next state control logic for $G_C$ is:

$$pc' \equiv \bigwedge_{i=1}^{n} c_i'.$$

### 4.2. Data logic

A data logic subgraph of $G$ is defined as $G_D = \langle B, X, A, \delta \rangle$. $G_D$ discards all the control flow information and concentrates on data updates at individual blocks. Fig. 3 shows the data logic subgraph for the program shown in Fig. 1.

Assume a variable $x_i \in X$ is assigned in blocks $b_{ij}$ $(1 \leq j \leq k)$ by expressions $expr_{ij}$ and not assigned in $b_{ij}$ $(k < j \leq n)$. The next state data logic for $x_i$ can be written as:

$$x_i' = \left( \bigvee_{j=1}^{k} (pc = index(b_{ij})) \wedge expr_{ij} \right) \vee \left( \bigvee_{j=k+1}^{n} (pc = index(b_{ij})) \wedge x_i \right),$$

where $index(b)$ returns the index value of block $b$. The next state control logic for $G_D$ is:

$$X' \equiv \bigwedge_{x_i \in X} x_i'.$$

In order to obtain the binary logic for each variable assignment $x' = expr$, we build a combinational circuit for $expr$. For example, to handle an expression of type $expr1 \& expr2$ (bit-wise AND), we first build circuits for the sub-expressions $expr1$ and $expr2$. Let vectors $vec1$ and $vec2$ be the outputs of these circuits. The final result has the same bit-width as $vec1$ and $vec2$, and each result bit is the output of an AND gate with two inputs being the corresponding bits in $vec1$ and $vec2$. To handle an expression of type $expr1 + expr2$, we create an $n$-bit adder. For the case of a relational expression the result has only one bit.

## 5. Model analysis and customized decision heuristics

The Boolean circuit models generated by our software modeling contain many common features that are based on the particular translation presented here. Although we model the program counter to track progress in the control flow graph, there is additional information in the original program that could improve the efficiency of the SAT-based BMC. Although each unrolling introduces the whole code into the satisfiability problem, many blocks in the new unrolling can be declared unreachable by merely considering the control flow of the software program. In the following, we propose various heuristics that improve the performance of the SAT-based BMC for the Boolean models generated from software programs. In Section 8, we show the effectiveness of these heuristics in our experiments.

### 5.1. Decision scoring of PC variables

We adjust the decision heuristics used by the SAT-solver in our VeriSol verification engine to take advantage of the fact that we are analyzing a Boolean circuit model generated from a software program. A simple decision heuristic that increases the likelihood that the SAT-solver makes decisions first on variables that correspond to the control flow rather than the data flow, takes advantage of the fact that each new unrolling does not allow the whole code to be reached.

We implement this heuristic by increasing the score for the bits of the program counter variable, which in turn makes the back-end SAT-solver choose these variables as decision variables first. In addition to scoring program counter variable bits higher than other variables in the system, we also control how to vary the scoring of these bits over various time-frames. For example, variables in later time-frames can be scored higher than those in the earlier time-frames.

### 5.2. One-hot encoding of basic blocks using selection bits

Another heuristic that we implemented is a one-hot encoding of the pc variables which allows the SAT-solver to make decisions on the full program counter variable instead of the individual bits of the program counter. In addition to the binary encoded program counter variable already present in the circuit, we add a new selection bit for each basic block. The selection bit is set iff the basic block is active, i.e. when a certain combination of program counter variable bits is valid. This provides a mechanism for word-level decisions. Furthermore, it increases the efficiency of the SAT solver, since a certain basic block selection bit automatically invalidates all other selection bits through the program counter variable bits.

By increasing the score of the selection bits in comparison to other variables in the system, we are able to influence the SAT-solver to make quick decisions on the location first. An obvious disadvantage to this heuristic is that we need to add one selection bit to the Boolean circuit model per basic block in the software code. However, in practice this disadvantage is mitigated by the performance improvements as described in Section 8.

In addition, we can increase the score for these basic block selection bits by considering the pre-computed static reachability information from the CFG. We increase the score for only those basic block selection bits at depth $k$, if the corresponding basic block can actually be reached statically in the CFG at depth $k$. This requires an additional BFS of reachable basic blocks at various depths in the CFG.

As an example, let $b_i$ represent the selection bit for basic block $i$ and let $c_i$, $0 \leq i \leq 3$ represent the four needed program counter variable bits for the code shown in Fig. 1. We add constraints for each basic block, asserting the equality of the block selection variable with the binary-encoded program counter label. For example, $b_0 = \bar{c_3}\bar{c_2}\bar{c_1}\bar{c_0}$ and $b_6 = \bar{c_3}c_2c_1\bar{c_0}$. Now, a decision by the SAT-solver to assign $b_0$ to 1 corresponds to a word-level decision on the program counter, which immediately results in implying $b_i = 0$, $i > 0$, including $b_6 = 0$ in our example.

Furthermore, similar to the previously described technique of scoring variables in later time-frames higher than in prior time-frames, we can also use this strategy for scoring of the selection bits.

### 5.3. Explicit modeling of incoming CFG transitions

We can also aid the analysis of the back-end SAT-solver by constraining its search space to eliminate impossible predecessor–successor basic block combinations in the CFG. These constraints capture additional high-level information, which helps to prune the search space of the SAT-solver. At each depth, the choice by the SAT-solver to consider a particular basic block enables a limited number of possible predecessor blocks and eliminates immediately all other basic blocks from consideration.

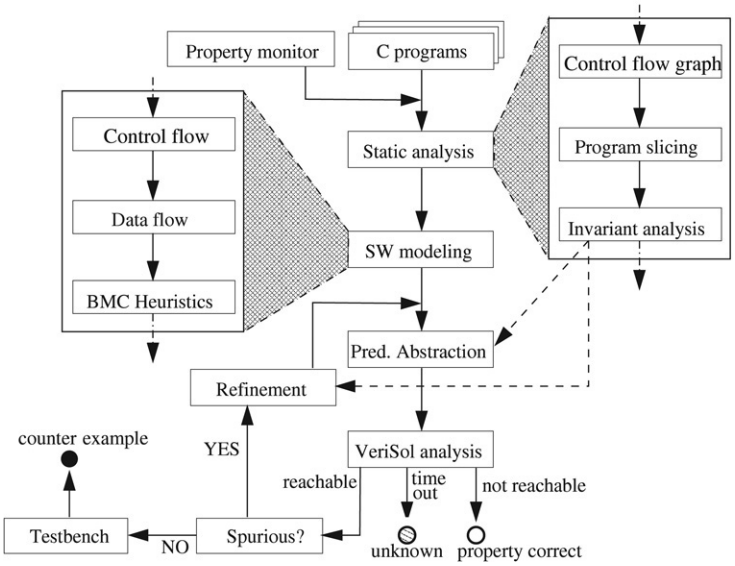


**Fig. 4.** F-SOFT tool overview.

By increasing the likelihood that the SAT-solver decides first on the program counter, we take advantage of the fact that each new unrolling does not allow all basic blocks to be reachable at each depth. We prefer to add these constraints based on incoming transitions into a basic block, since the Boolean model of the software already encodes the outgoing transitions in terms of the program counter variable. This helps the SAT-solver since it provides bi-directional information about the transitions in the CFG.

For example, let $c'_i$, $0 \leq i \leq 3$, denote the next state program counter variable bits in Fig. 1. The following constraint is added for basic block 7: $\bar{c'_3}\bar{c'_2}c'_1c'_0 \rightarrow (\bar{c_3}\bar{c_2}c_1c_0 \vee \bar{c_3}c_2\bar{c_1}\bar{c_0})$, to reflect the constraint that if the current block is 7, then the predecessor block is either 3 or 4. Similarly, for basic block 3, we add the following constraint: $\bar{c'_3}\bar{c'_2}c'_1c'_0 \rightarrow (\bar{c_3}\bar{c_2}\bar{c_1}c_0 \wedge g)$, where $g$ stands for the representation of x>4. We can also add such constraints in terms of the one-hot encoded selection bits for the basic blocks if both these heuristics are being utilized. The constraint for basic block 7 then simply becomes $b'_7 \rightarrow (b_3 \vee b_4)$.

We also allow the user to customize this heuristic by choosing a subset of basic blocks for which to apply this heuristic. This customization allows the user to fine-tune the amount of additional constraints generated, since too many additional constraints may burden the SAT-solver rather than improve performance.

In addition, another optimization, that we have not yet implemented, is to limit the additional constraints at each depth to those basic blocks that are actually reachable at that depth. This optimization, similar to the optimization described in the one-hot encoding heuristic, requires an additional BFS of reachable basic blocks.

## 6. Tool overview

In this section we describe our verification tool F-SOFT [40]. Its novelty lies in the combination of several recent advances in formal methods research including SAT-based verification, static program analyses and predicate abstraction. As shown in Fig. 4, we translate a program into a Boolean representation to be analyzed by our verification engine VERISOL (based on DIVER) [26] which includes BDD- and SAT-based model checking techniques. In addition to these standard model checking techniques, it also contains specialized model checking engines that are targeted for software models. These include use of disjunctive BDDs [69], and use of multiple symbolic representations using a combination of decision procedures for Boolean and integer representations [70].

The tools also includes various static program analyses, such as computing the control flow graph of the program, performing program slicing with respect to the property and performing range analysis as briefly described in the following. We model the software using a Boolean circuit representation, including customized heuristics for analysis by a SAT-solver, as described earlier in Section 5. We perform a localized predicate abstraction with register sharing with support of statically computed numeric invariants, if the user so chooses.

The actual analysis of the resulting Boolean circuit model is then performed using the VERISOL tool. If a counterexample is discovered, we use a testbench generator that outputs an executable program for the user to analyze the bug in his/her favorite debugger. In the following, we briefly describe various stages of the F-SOFT tool.

### 6.1. Static program analysis

As mentioned before, F-Soft relies on a variety of static program analysis steps to reduce the model of the program and number of properties as much as possible before model checking is performed on the remaining model and properties. We briefly touch upon a few of the novel techniques that are employed in F-Soft.

**Range Analysis.** Since model checking suffers from the state explosion problem which is further exacerbated in the context of software verification, we introduce the use of lightweight, efficient and sound abstraction techniques to statically determine possible ranges for values of program variables. Such range information for each variable can be used to improve the efficiency of software verification tools by providing a smaller state-space to be considered by the back-end model checking engines such as those based on BDDs or SAT-solvers.

In addition, range information can be used to reduce the size of bit-blasted verification models to which programs are translated for verification purposes: a 3-bit adder can be expressed with fewer Boolean operators than a 32-bit adder. Furthermore, such range information can be used to improve the efficiency of other popular techniques such as predicate abstraction and counterexample-guided abstraction refinement as described in [71]. Although range analysis techniques have been used for other applications [58], we believe we are the first to use them for software model checking.

**Invariant analysis using numeric domains.** Numerical domain static analysis has been used to prove safety of programs for properties such as the absence of buffer overflows, null pointer dereferences, division by zero, string usage and floating point errors [7,23,68]. Domains such as *ranges* (or intervals), *octagons*, and *polyhedra* are used to symbolically over-approximate the set of possible values of integer and real-valued program variables along with their relationships under the *abstract interpretation framework* [12,19,21,29,43,53,59]. These domains are classified by their *precision*, i.e, their ability to represent sets of states, and *tractability*, the complexity of common operations such as union (join), post-condition, widening and so on. In general, enhanced precision leads to more proofs and less false positives, while resulting in a costlier analysis.

Fortunately, applications require a domain that is "*precise enough*" rather than "*most precise*". As a result, research in static analysis has resulted in numerous trade-offs between precision and tractability. The octagon abstract domain, for instance, uses polyhedra with two variables per constraint and unit coefficients [53]. The restriction yields fast, polynomial time domain operations. Simultaneously, the pair-wise comparisons captured by octagons also express and prove many common run time safety issues in practical software [7]. Nevertheless, a drawback of the octagon domain is its inability to reason with properties that may need constraints of a more complex form. We have incorporated various inter-procedural invariant analysis engines into the F-Soft tool that are based on disjunctive analysis [61] and a new numeric domain called *symbolic range domain* [60].

### 6.2. Predicate abstraction

Predicate abstraction is a popular technique for extracting finite state models from software [4]. 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 too many predicate relationships.

Our contribution [41] is inspired by the lazy abstraction approach and the localization techniques implemented in BLAST [35]. BLAST makes use of interpolation, whereas we use weakest pre-conditions to find predicates relevant at each program location. While weakest pre-conditions have been used for predicate abstraction techniques before, we are the first to use them to find predicates in a localized (block-based) fashion.

The performance of BDD-based model checkers depends crucially on the number of state variables. Due to predicate localization most predicates are useful only in certain parts of the program. The state variables corresponding to these predicates can be *shared* to represent different predicates in other parts of the abstraction. However, this might result in too many abstraction refinement iterations; e.g., if the value of a certain predicate needs to be tracked at multiple program locations. Since we add predicates lazily only along infeasible traces, the fact that a predicate is globally useful will be learned only through multiple abstraction refinement iterations. We make use of a simple heuristic for deciding when the value of a certain predicate may need to be tracked globally or in a complete functional scope. If the value of a predicate needs to be tracked in a large scope, then it is assigned a *dedicated* state variable which is not reused for representing the value of other predicates in the same scope. More details on our localized predicate abstraction approach using register sharing can be found in [41].

We further improve the predicate abstraction refinement loop through the use of a priori statically computed invariants. The main idea is to selectively strengthen (conjoin) the concrete transition relation at a given program location by efficiently computed invariants that hold at that program location. We observe that the abstract models produced by predicate abstraction of strengthened transition relation are more precise leading to fewer spurious counterexamples, thus, decreasing the total number of abstraction refinement iterations. Furthermore, the length of relevant fragments of spurious traces needing refinement shortens. This leads to an addition of fewer predicates for refinement. Additional details on the use of invariants for predicate abstraction can be found in [42].

### 6.3. Testbench generation and debugging

The back-end verification in F-Soft is performed by the VeriSol tool. If VeriSol discovers a real counterexample, F-Soft reports to the user a descriptive one-line summary of the bug. For example, consider the source code for function `pointer2`

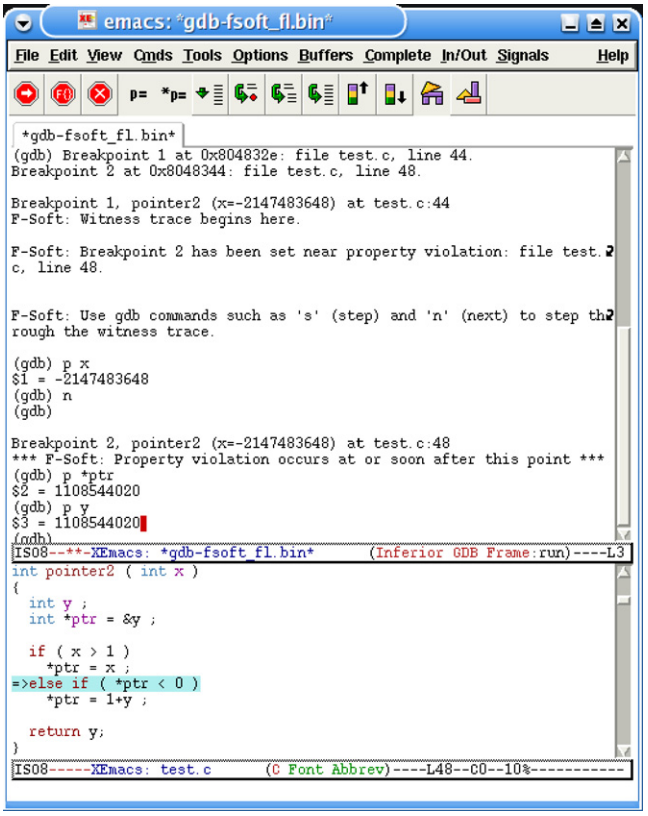


**Fig. 5.** F-SOFT testbench generator for `gdb` in `emacs`.

in the lower half of Fig. 5. In this example, F-SOFT performed the check for the use of uninitialized variables. Here, F-SOFT reports that a bug was found in file test.c at line 48: At least one variable not initialized in a condition.

In addition to the bug summary, F-SOFT generates an error trace on the original source code, to help the user in debugging. This error trace is also utilized by an automatic testbench generator, which generates an executable program for the user to analyze the bug in more detail in his/her favorite debugger. Fig. 5 shows this capability using the `emacs` front-end to the commonly used `gdb` debugger. In effect, the F-SOFT testbench executable initializes the memory state according to the discovered counterexample, and automatically sets breakpoints in the source code at places that are deemed interesting for demonstrating the bug. Furthermore, F-SOFT adds descriptive messages into the output of `gdb`, thus improving the users' understanding of the bug. More importantly, the testbench allows the user to debug in a familiar environment, utilizing all features provided by standard debuggers, such as inspection of data, setting of breakpoints etc. The output in the upper half of Fig. 5 shows a sample execution of our generated testbench. Note that external variables and parameters (here: parameter `x`) are initialized according to the discovered bug, and breakpoints are automatically set at the entry to the main function for analysis purposes `pointer2` and near the line where the bug occurs. Note also the use of data inspections using `gdb`'s `p` (display) command, and tracing using the `n` (next) command.

## 7. Network case study: Point-to-point protocol

The verification of various network protocols has been a popular application of model checkers [24]. Most of these approaches have analyzed models of network protocols with respect to some temporal properties such as absence of deadlock where the model is based on a textbook description or an RFC (Request for Comments) document. Recently, there have also been attempts to verify an implementation of a protocol with respect to its standard defined in an RFC [3]. In this section, we describe the Point-to-Point protocol, which is one of the case studies for F-SOFT.

The Point-to-Point Protocol (PPP) provides a standard method for transporting multi-protocol datagrams over point-to-point links. PPP contains three main components, namely a method for encapsulating multi-protocol datagrams, a Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection, and a family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols. Similar to [3], we focus on checking the implementation of the option negotiation automaton of the LCP part of PPP for link establishment.

RFC 1661 [64] specifies the complete state transition table of the automaton. It is a finite state machine with 10 states, which reacts to 15 events. The automaton can switch states when receiving an event, and also perform other actions, such

**Table 2**
A part of the PPP specification

| | Stopped | Req-Sent | Opened |
|---|---|---|---|
| Close | goto Closed | Term-Req<br>goto Closing | Term-Req<br>goto Closing |
| RCA | Term-Ack | goto Ack-Rcvd | goto Req-Sent |
| RTR | Term-Ack | Term-Ack | Term-Ack<br>goto Stopping |
| RTA | | | Conf-Req<br>goto Req-Sent |

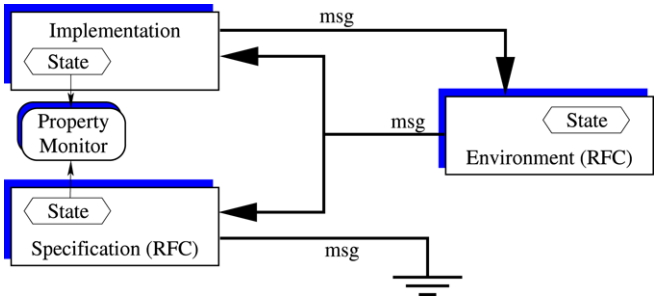


**Fig. 6.** PPP verification setup.

as sending replies. Any implementation of the PPP has to follow the behavior described in the RFC, which is partly shown in Table 2 for the states `Stopped, Req-Sent` and `Opened`. We only present the information about which messages should be sent back, if any, and what the next state should be if there is a change of states. An empty field describes the fact that the automaton will simply ignore a received packet.

We consider an open-source implementation of the protocol (`ppp-2.4.0`) distributed in various Linux systems. In this paper we assume that events and actions are handled correctly by the implementation. The following represents a code fragment of the public implementation:

```
static void fsm_rtermack(f)
    fsm *f;
{
    switch (f->state) {
    /*NOTE: other cases removed for brevity*/
    case OPENED:
        if (f->callbacks->down)  /*Inform upper layers*/
            (*f->callbacks->down)(f);
        fsm_sconfreq(f, 0);
        break;
    }
}
```

For the purpose of this paper, we edit the public implementation to be limited to LCP. We want to verify that the public implementation adheres to the specification as given in RFC 1661. The prototype implementation did not support verification of multi-threaded code at the time of these experiments [39]. We implemented the following scheme to analyze the implementation of PPP which is illustrated in Fig. 6: To check the public implementation we need a second peer to engage it. We developed an environment peer based on the RFC in C, that also initiates requests non-deterministically. Since our goal is to analyze whether the public implementation adheres fully to the RFC specification, we also implemented a specification model of the RFC in C.

The environment and the public implementation are engaged in communication, while the specification "listens in" on messages received by the public implementation. Although the specification tries to respond to these messages, its outgoing link is "grounded" and does not affect the environment peer. Our analysis thus translates to the question whether the state of the public implementation always matches the state of the RFC-based specification starting from such safe and correct states.

In [3], the C-program as described here, has been manually translated to the input language of the model checker Mocha[2]. In contrast to that approach, we actually perform model checking based on a slightly modified original source code after providing an appropriate environment for the property at hand. The analysis in [3] shows that the public implementation does not fully adhere to the specification given by RFC 1661. When the peer receives a packet `RTA`, it is supposed to send back a configuration request, which is implemented correctly in the public implementation. However, it

**Table 3**
Experimental results

| | Latches | Gates | In | std | sc | 1h | tr | sc&1h | sc&tr | 1h&tr | All |
|---|---|---|---|---|---|---|---|---|---|---|---|
| W1 | 62/1314 | 22k–24k | 360 | TO | TO | 1052 | TO | **1035** | TO | 1048 | 1040 |
| W2 | 68/1311 | 20k–28k | 30 | TO | TO | 302 | TO | **300** | TO | 302 | **300** |
| W3 | 209/1114 | 14k–19k | 30 | 5706 | 5537 | 2240 | TO | 2235 | 5619 | **2189** | 2240 |
| W4 | 406/709 | 5210–6426 | 0 | TO | TO | 1463 | TO | 1485 | TO | 1462 | **1460** |
| W5 | 444/747 | 5731–7250 | 0 | TO | TO | 2933 | 6511 | 2935 | 7448 | **2930** | 2946 |
| P1 | 18/50 | 459–621 | 6 | 1714 | 1692 | 1678 | 1198 | 1688 | **1043** | 1694 | 1747 |
| P2 | 18/50 | 459–621 | 6 | 4087 | 5228 | 8289 | **3442** | 8856 | 4006 | 6903 | 6988 |
| P3 | 55/123 | 785–1065 | 15 | **194** | 205 | 255 | 197 | 255 | 215 | 255 | 255 |

is also supposed to update its internal state to `Req-Sent`, which is missing in the implementation. We present experimental results for this case study in more detail in the following section.

## 8. Experimental results

We have performed various case studies comparing the customized decision heuristics using F-Soft. In the following we describe the results of multiple case studies that we have recently completed. In most experiments we assume a 4 h time limit for the analysis. For the first five case studies named W1–W5 the analysis finds that the stated property is incorrect and a *witness* trace exists, while for the later three case studies named P1–P3 the analysis finds a *proof* for the property.

We summarize the experimental results in Table 3. W3 is the PPP network protocol case study described in Section 7, while W1 and W2 are related pieces of code with different complexities. The designs W4 and W5 are based on a different example — they have the same source code but consider different properties. The code for W4 and W5 is relatively small and is purely straight-line code with no loops. The provable case studies P1 and P2 are based on a simple reactive and interleaved mutual exclusion example where we use different proof strategies to derive the correctness proof. P3 is an unrelated simple code fragment based on interface specification checking.

### 8.1. Results for customized heuristics used in BMC

For each case study, Table 3 shows the number of latches in the Boolean circuit representation in the second column. We include two numbers, where the first number corresponds to the number of latches where the one-hot encoding heuristic is not used, while the second number represents the number of latches with one-hot encoding. Please note that we uses the range analysis techniques [58,71] to limit the number of bits per variable. We also compute the cone of influence of the circuit model with respect to the property which further reduces the number of latches in the model.

The next column labeled **gates** contains the number of gates in the Boolean circuit models for these case studies, where the range depends on the subset of heuristics used for the various verification runs. The lower bound corresponds to the case that no customized heuristics are used, while the upper bound represents the number of gates in the case that all heuristics are used. The following column labeled **in** denotes the number of primary inputs needed for the analysis of the respective property.

The following columns represent the experimental results with respect to CPU time measured in seconds for our BMC analyses using the various combinations of heuristics. The column labeled **std** represents the case when we use standard VeriSol without any of the customized heuristics described here. It can be seen that the standard VeriSol implementation is not able to complete the analysis of four of the eight models due to a time-out, denoted TO, where the 4 h time-limit expires.

The remaining columns show the experimental results for various combinations of heuristics used for the analysis. We abbreviate the heuristics *scoring of pc variables* by **sc**, *one-hot encoding* by **1h** and *explicit modeling of CFG transitions* by **tr**. For example, the column **1h&tr** corresponds to the combination of one-hot encoding and explicit modeling of CFG transitions, while **all** includes additionally the scoring of pc variables.

The data show that using just the one-hot encoding heuristic proves very efficient for the analysis of models W1–W5 when compared to the standard VeriSol heuristics, while the respective results for models P1–P3 are somewhat mixed. Using the scoring pc variables heuristic by itself, on the other hand, does not seem to benefit our analysis in this set of experiments. In contrast to the one-hot encoding heuristic, it is interesting to note that the explicit modeling of CFG transitions seems to be helpful in models P1–P3, while ineffective for models W1–W5 when used individually.

When considering the possible combinations of heuristics, it is worth noting that for models W1–W5 the advantage of using the one-hot encoding is supported by the addition of other heuristics. A similar statement can however not be made for the provable models P1–P3 where the results are currently inconclusive with respect to the best combination of heuristics. For P1 we find that using scoring of pc variables with explicit modeling of CFG transitions improves the performance somewhat over the case that only the explicit modeling of transitions is used.

In summary, we find that in most cases a combination of one-hot encoding of basic blocks with explicit modeling of incoming transitions is a good default choice when the BMC searches for both a proof and a witness. When using BMC purely to find bugs, using one-hot encoding only seems quite effective.

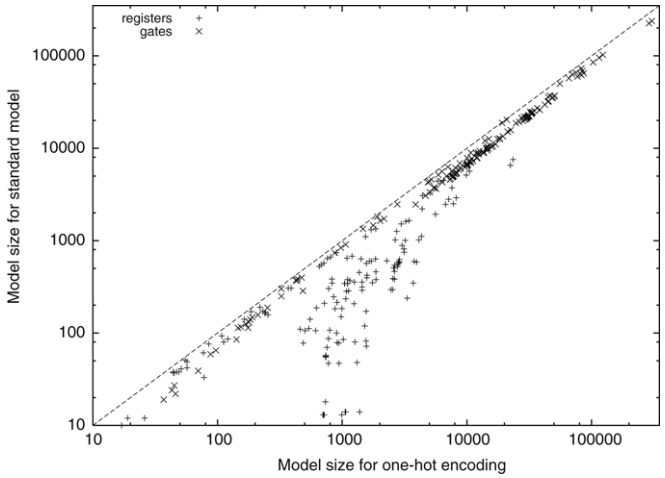


**Fig. 7.** Model size comparison of BMC heuristics for standard property checkers.

### 8.2. Customized BMC heuristics for standard property checkers

We have also performed experiments with the customized BMC heuristics for a proprietary software package, where we analyzed the source code for various standard property violations using the F-Soft tool [38]. We analyzed the source code for array bound violations, use of uninitialized variables, and certain locking rules violations. In total, we analyzed 177 distinct, *non-trivial* cases.[1] The code contains over 9400 lines of C code after some preprocessing steps. The experimental results are summarized in Figs. 7 and 8.

First consider Fig. 7, which presents a comparison of the generated models for the various checks with and without the one-hot encoding heuristics discussed in Section 5. The figure contains a scatter plot on a logarithmic scale of the number of registers required with and without the one-hot encoding of basic blocks (+), and the corresponding number of Boolean connectives (×). As expected, all data points are at or below the diagonal, which means that in all cases one-hot encoding introduces additional registers and additional Boolean connectives. More interestingly, it can be seen graphically that the number of registers using one-hot encoding can substantially increase the complexity of the model, while the number of Boolean connectives remains relatively unaffected. In these experiments, we found a tripling of the number of required registers, but only 32% additional Boolean connectives on average.

Consider now Fig. 8, which shows the comparison of the actual verification run-time (in seconds) observed on the models shown in Fig. 7, on a logarithmic scale. Any point below the diagonal corresponds to a case where the verification was faster without the additional one-hot constraints, while points above the diagonal denote cases where the additional constraints improved the run-time. Note that time-outs are denoted as 6000 s of run-time in the scatter plot.

As can be seen, there are many cases where the additional constraints are not useful, and the standard model was able to complete the analysis in slightly less time. On average, for cases that were completed by both methods, we were able to see a 3% increase in run-time by adding the one-hot constraints. Note that this increase is relatively small compared to the increase in the model sizes shown in Fig. 7. We believe that this is largely due to the SAT-solver heuristics which allow the SAT search to focus on relevant variables without getting unduly affected by the total number of state variables. It should also be noted that as verification complexity increased (as measured by longer run-times), the one-hot encoded models were increasingly outperforming the models generated without the additional constraints. In particular, we found three cases out of the 177 non-trivial checks that timed out using the standard model, while we were able to complete their analysis using the additional one-hot encoding.

If we divide the cases into two groups that correspond to the cases above the diagonal and below the diagonal we further see the improvements gained by one-hot encoding for harder cases. In cases where the standard model outperformed the one-hot encoded model we found that it took about 10.34 s for the standard model and 13.95 s for the one-hot encoded models, which corresponds to a run-time penalty of around 35%. We believe that for most of these cases the standard properties were relatively easy to analyze. On the other hand, for models that were analyzed faster using one-hot encoding and were completed by both methods, we found that the average time was 40.74 s for one-hot encoded models, but 65.06 s for standard models, which corresponds to a 60% faster performance with one-hot encoded models. Furthermore, it should be emphasized again that verification for three models was completed with one-hot encoding but could not be completed within the time bound using the standard models.

[1] By non-trivial cases we mean that the respective property monitors were statically able to fire a violation in the control flow graph of the considered program piece.

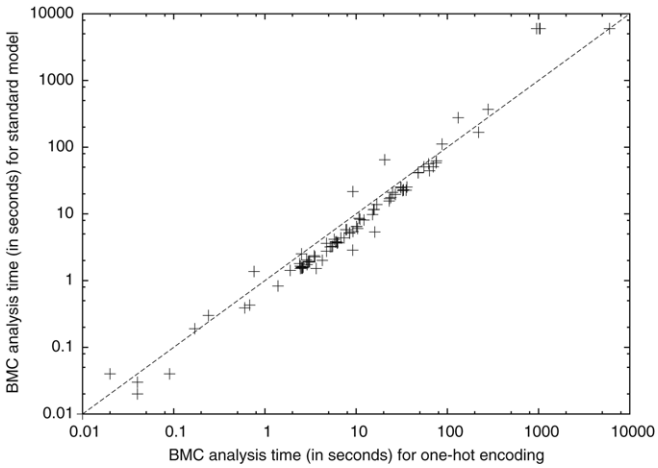


**Fig. 8.** Run-time comparison of BMC heuristics for standard property checkers.

### 8.3. Results using BDDs

In addition to the results presented in Table 3, we have also used our standard BDD-based model checker on the generated models. For designs W1, W2, W4 and W5 our BDD-based model checker was able to find a witness trace, albeit being slower than BMC and taking up to four times the amount of CPU time. In the case of W3 (the PPP case study), the BDD-based model checker was not able to complete the analysis in the given time-bound.

For designs P1–P3, the BDD-based model checker beat any of the BMC-based analyses given in Table 3 due to the small model sizes. It should also be noted that we have been able to improve upon the BDD engines used in these experiments by performing a disjunctive image computation utilizing the basic block information even for BDD-based solvers [69]. However, our experience in hardware verification suggests that SAT-based proof techniques become increasingly important compared to BDDs when considering larger provable models. Our current results provide preliminary evidence for the effectiveness of our customized SAT heuristics.

### 8.4. Basic blocks vs. individual statements

We have also performed experiments to compare our software modeling framework using basic blocks to an analysis based on individual statements, as used in CBMC. Even for a small example (W4), we found that a BMC analysis using the one-hot encoding heuristic took about 25% more time for the statement analysis than our implementation that uses basic blocks. The length of the witness trace increased from 151 to 187, which corresponds to an increase of 24%. Therefore, the final SAT problem size generated by the BMC unrolling for the case of individual statements increased by about 27%. This is due to the fact that we have a 24% increase of the witness length, where each individual unrolling is about 11% larger than that of basic blocks. We believe that our approach using basic blocks will be even more effective for larger case studies.

We performed additional related experiments on a variety of benchmarks checking for array bound violations, NULL pointer dereferencing and C string library misuse. We show the advantage of performing additional parallelization of the CFG after various static program simplifications have performed manipulations of the CFG. These simplifications include amongst other things program slicing, constant propagation, and invariant analysis using numerical domains [60,61]. After these CFG manipulations, we can *merge* basic blocks by performing additional parallelization steps. This merging step compresses an uninterrupted sequence of basic blocks into a single basic block thus potentially shortening path lengths as well as avoiding intermediate temporary variables.

We present experimental results of the post-simplification merging of basic blocks in Table 4. The table includes several benchmarks, where we checked a variety of properties for multiple API-level entry points per benchmark. We report the sum over all API-level entry points in the table. For each benchmark we report the *relevant LOC*, and the total number of properties that is being analyzed by the tool for the various API-level entry points. Note that the number of properties depends both on the size of the code and the type of check that is being performed. Then, we report the statistics for the flow where the simplifications are performed, but no additional merging of basic blocks is happening after these simplifications. We provide the information on how many variables (**Vars**) are still left in the final model that is given to the model checker, as well as how many basic blocks (**Blks**) remain in that model. We also describe how many proofs (**Prf**) are reported by our tool (which includes both proofs by static analysis and proofs by the model checker), as well as how many witnesses (**Wit**) are discovered, and how many properties have not been resolved due to time-outs (**T/O**). Finally, we report the same information for the post-simplification-merging-based approach running with the same time-out.

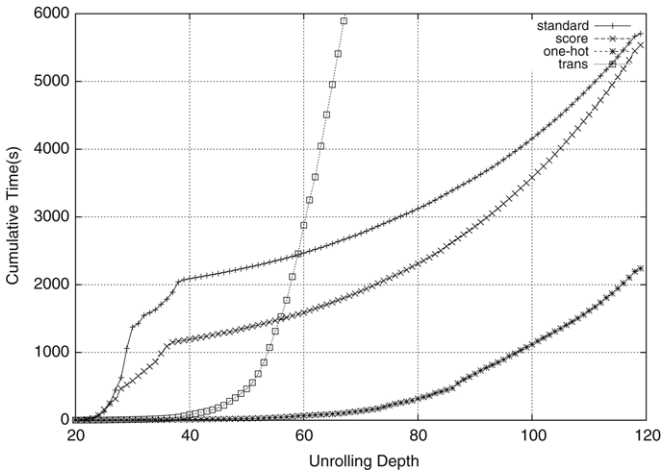


**Fig. 9.** Cumulative time comparison of BMC heuristics for design W3.

**Table 4**
Experimental results on post-simplification merging

| | LOC | Properties | Without merging | | | | | Merging blocks | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Vars | Blks | Prf | Wit | T/O | Vars | Blks | Prf | Wit | T/O |
| D1 | 8.3k | 628 | 957 | 1723 | 249 | 11 | 368 | 663 | 1198 | 268 | 25 | 335 |
| D2 | 5.9k | 350 | 445 | 718 | 247 | 54 | 49 | 368 | 521 | 245 | 65 | 40 |
| D3 | 5.5k | 173 | 295 | 557 | 128 | 37 | 8 | 237 | 372 | 128 | 37 | 8 |
| D4 | 7.7k | 116 | 538 | 750 | 96 | 11 | 9 | 401 | 391 | 104 | 11 | 1 |
| D5 | 15.4k | 1005 | 726 | 1776 | 1000 | 3 | 2 | 563 | 1224 | 1001 | 3 | 1 |
| D6 | 28.4k | 39 | 2269 | 1048 | 35 | 4 | 0 | 2262 | 739 | 35 | 4 | 0 |
| D7 | 21.4k | 961 | 2300 | 1876 | 768 | 83 | 110 | 2064 | 1128 | 771 | 86 | 104 |
| D8 | 21.5k | 484 | 1132 | 1565 | 387 | 55 | 42 | 953 | 886 | 393 | 55 | 36 |
| **Total** | 114k | 3756 | 8662 | 10013 | 2910 | 258 | 588 | 7511 | 6459 | 2945 | 286 | 525 |

As can be seen in Table 4, we are able to increase the number of found witnesses by 11% by using post-simplification merging. We also increase the number of proofs slightly,[2] and therefore decrease the number of properties with unknown answer by 11%. These gains are partly explained by the decrease in the final model size that is passed to the model checker, which contains about 13% less variables, and 35% less basic blocks.

### *8.5. Detailed results for PPP*

Figs. 9 and 10 show a more detailed comparison of the individual heuristics and their performance for the PPP case study (W3). Fig. 9 shows the cumulative time in seconds taken for all depths up to a given number, while Fig. 10 shows the respective time needed for a particular depth.

The graphs labeled *standard* represent the standard decision heuristics implemented in the VeriSol tool. The graph shows the advantage of using a SAT-based BMC for the analysis, since the standard includes various peaks in the computation time, in particular for depths 20–35, but better performance afterwards. This indicates that the SAT-solver is able to learn important invariants of the design early on that enable a deeper analysis later.

The figures also include three more graphs each, describing the respective performance of BMC using the heuristics scoring of pc variables (*score*), one-hot encoding (*one-hot*) and encoding of CFG transitions (*trans*) individually. It is worth noting that the inherent learning in the SAT-solver is preserved, which is visible by the various peaks in the respective graphs. Fig. 9, in particular, shows the advantage of the one-hot encoding heuristic for the design W3 which consistently outperforms the other heuristics.

## 9. Conclusions and future work

This paper describes the customized SAT-based heuristics we have used for efficient analysis of models generated by our software verification methodology. This methodology is founded upon basic block software modeling, translating a C program into a Boolean representation. The resulting model is analyzed by a back-end SAT-based or BDD-based

[2] It should be noted that we increased the number of proofs found during the model checking stage by 17%.

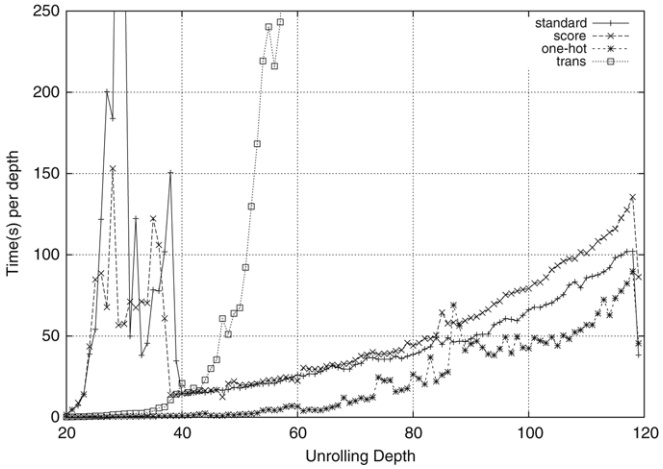


**Fig. 10.** Time-per-depth comparison of BMC heuristics for design W3.

bounded or unbounded model checker. It is applicable for a large subset of the C programming language allowing pointer arithmetic and bounded recursion. We consider reachability properties, in particular whether certain assertions or basic blocks are reachable in the source code, or whether certain standard property violations can occur. This paper also described the implementation of these as part of our F-Soft tool [40]. We also presented detailed experimental results using our customized heuristics for various case studies.

There remain many research directions to follow in the future. We need to experiment with the heuristics presented in this paper more and apply them on larger case studies as well. Additional larger case studies may point us to consider other heuristics in the SAT solver also. We want to further investigate the potential synergies between static program analysis and software model checking. Furthermore, we are continuing our efforts in improving the efficiency of our predicate abstraction implementation.

## Acknowledgements

We thank Srihari Cadambi, Himanshu Jain, Vineet Kahlon, Weihong Li, Nadia Papakonstantinou, Sriram Sankaranarayanan, Ilya Shlyakhter, Chao Wang and Aleksandr Zaks for their help in the development of the F-Soft tool as described in Section 6.

## References

[1] R. Alur, C. Courcoubetis, D.L. Dill, Model-checking in dense real-time, Information and Computation 104 (1) (1993) 2–34.
[2] R. Alur, L. deAlfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, B.Y. Wang, MOCHA: A model checking tool that exploits design structure, in: Intern. Conf. on Software Engineering, 2001, pp. 835–836.
[3] R. Alur, B.-Y. Wang, Verifying network protocol implementations by symbolic refinement checking, in: 13th Conference on Computer Aided Verification, 2001, pp. 169–181.
[4] T. Ball, R. Majumdar, T.D. Millstein, S.K. Rajamani, Automatic predicate abstraction of C programs, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001, pp. 203–213.
[5] T. Ball, S. Rajamani, Bebop: A symbolic model checker for Boolean programs, in: SPIN 2000 Workshop on Model Checking of Software, in: LNCS, vol. 1885, Springer, 2000, pp. 113–130.
[6] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: Proceedings of the 36th ACM/IEEE Design Automation Conference, 1999, pp. 317–320.
[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A Static Analyzer for Large Safety-Critical Software, in: ACM SIGPLAN PLDI'03, vol. 548030, ACM Press, June 2003, pp. 196–207.
[8] A. Boujjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: Applications to model checking, in: CONCUR'97: Concurrency Theory, Eighth International Conference, in: LNCS, vol. 1243, Springer, 1997, pp. 135–150.
[9] M. Browne, E.M. Clarke, O. Grumberg, Reasoning about networks with many identical finite state processes, Information and Computation 81 (1989) 13–31.
[10] R.E. Bryant, Graph-based algorithms for boolean-function manipulation, IEEE Transactions on Computers C(35) (1986) 677–691.
[11] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, D. Wang, Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis, in: M. Aagaard, J.W. O'Leary (Eds.), Proc. of the 4th Conference on Formal Methods in Computer-Aided Design, FMCAD, in: LNCS, vol. 2517, November 2002, pp. 33–51.
[12] R. Clarisó, J. Cortadella, The octahedron abstract domain, in: Static Analysis Symposium, in: LNCS, vol. 3148, Springer-Verlag, 2004, pp. 312–327.
[13] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, T. Teitelbaum, Programs slicing for VHDL, International Journal on Software Tools for Technology Transfer (STTT) 4 (1) (2002) 125–137.
[14] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, Computer Aided Verification (2000) 154–169.
[15] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 2000.
[16] E.M. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: K. Jensen, A. Podelski (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 2988, Springer, 2004, pp. 168–176.
[17] E.M. Clarke, D. Kroening, N. Sharygina, K. Yorav, Predicate abstraction of ANSI-C programs using SAT, in: Model Checking for Dependable Software-Intensive Systems Workshop, 2003.

[18] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, H. Zheng, Bandera: Extracting finite-state models from Java source code, in: International Conference on Software Engineering, 2000, pp. 439–448.
[19] P Cousot and, R Cousot, Static determination of dynamic properties of programs, in: Proceedings of the Second International Symposium on Programming, Dunod, 1976, pp. 106–130.
[20] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[21] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among the variables of a program, in: ACM POPL, January 1978, pp. 84–97.
[22] S. Das, D. Dill, S. Park, Experience with predicate abstraction, in: Computer Aided Verification, in: LNCS, vol. 1633, Springer, 1999, pp. 160–171.
[23] N. Dor, M. Rodeh, M. Sagiv, CSSV: Towards a realistic tool for statically detecting all buffer overflows in C, in: Proc. PLDI'03, ACM Press, 2003.
[24] J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu, CADP: A protocol validation and verification toolbox, in: Proc. of Intern. Conf. on Computer-Aided Verification, in: LNCS, vol. 1102, Springer, 1996.
[25] M.K. Ganai, A. Gupta, P. Ashar, Efficient memory modeling for SAT-based BMC, in: Computer Aided Verification, in: Lecture Notes in Computer Science, Springer, 2004.
[26] M.K. Ganai, A. Gupta, P. Ashar, DiVer: SAT-based model checking platform for verifying large scale systems, in: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 3340, Springer, 2005.
[27] P. Godefroid, VeriSoft: A tool for the automatic analysis of concurrent reactive software, in: Proceedings of the Ninth International Conference on Computer-Aided Verification, in: LNCS, vol. 1254, Springer, 1997.
[28] E. Goldberg, Y. Novikov, Verification of proofs of unsatisfiability for CNF formulas, in: Conference on Design Automation and Test in Europe, 2003.
[29] E. Goubault, S. Putot, Static analysis of numerical algorithms, in: SAS, in: LNCS, vol. 4134, 2006, pp. 18–34.
[30] S. Graf, H. Saidi, Construction of abstract state graphs with PVS, in: Computer Aided Verification, in: LNCS, vol. 1254, Springer, 1997, pp. 72–83.
[31] A. Gupta, M.K. Ganai, C. Wang, Z. Yang, P. Ashar, Abstraction and BDDs Complement SAT-based BMC in DiVer, in: W.A. Hunt Jr., F. Somenzi (Eds.), Proc. of the 15th Conference on Computer-Aided Verification, CAV, in: LNCS, vol. 2725, Springer, July 2003, pp. 206–209.
[32] A. Gupta, M.K. Ganai, Z. Yang, P. Ashar, Iterative abstraction using SAT-based BMC with proof analysis, in: Intern. Conf. on Computer Aided Design, November 2003, pp. 416–423.
[33] A. Gupta, Z. Yang, P. Ashar, A. Gupta, SAT-based image computation with applications in reachability analysis, in: Proceedings of the Third International Workshop on Formal Methods in Computer-Aided Design, in: LNCS, vol. 1954, Springer, 2000, pp. 354–371.
[34] K. Havelund, T. Pressburger, Model checking Java programs using Java PathFinder, Software Tools for Technology Transfer 2 (4) (2000).
[35] T.A. Henzinger, R. Jhala, R. Majumdar, K. McMillan, Abstractions from proofs, in: Symposium on Principles of Programming Languages, ACM Press, 2004, pp. 232–244.
[36] M. Hind, Pointer analysis: Haven't we solved this problem yet?, in: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'01, June 2001.
[37] G.J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[38] F. Ivančić, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, C. Wang, Z. Yang, Model checking C programs with F-SOFT, in: Intern. Conf. on Computer Design, ICCD, IEEE, 2005.
[39] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, P. Ashar, Efficient SAT-based bounded model checking for software verification, in: Symposium on Leveraging Formal Methods in Applications, ISoLA, 2004.
[40] F. Ivančić, Z. Yang, I. Shlyakhter, M.K. Ganai, A. Gupta, P. Ashar, F-SOFT: Software verification platform, in: Computer-Aided Verification, in: LNCS, Springer, 2005.
[41] H. Jain, F. Ivančić, A. Gupta, M.K. Ganai, Localization and register sharing for predicate abstraction, in: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 3340, Springer, 2005, pp. 397–412.
[42] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, C. Wang, Using statically computed invariants inside the predicate abstraction and refinement loop, in: T. Ball, R.B. Jones (Eds.), 18th International Conference on Computer Aided Verification, CAV, in: LNCS, vol. 4144, Springer, 2006, pp. 137–151.
[43] M. Karr, Affine relationships among variables of a program, Acta Informatica 6 (1976) 133–151.
[44] J. Krinke, Context-sensitive slicing of concurrent programs, in: 9th European Software Engineering Conference, ACM Press, 2003, pp. 178–187.
[45] D. Kroening, E.M. Clarke, K. Yorav, Behavioral consistency of C and Verilog programs using bounded model checking, in: Design Automation Conference, ACM Press, 2003, pp. 368–371.
[46] R.P. Kurshan, Computer-Aided Verification of Co-ordinating Processes: The Automata Theoretic Approach, Princeton University Press, 1994.
[47] S.K. Lahiri, R.E. Bryant, B. Cook, A symbolic approach to predicate abstraction, in: Computer Aided Verification, in: LNCS, vol. 2725, Springer, 2003.
[48] J.P. Marques-Silva, K.A. Sakallah, GRASP: A search algorithm for propositional satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.
[49] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer Academic Publishers, 1993.
[50] K.L. McMillan, Applying SAT methods in unbounded symbolic model checking, in: Computer Aided Verification, in: LNCS, vol. 2404, Springer, 2002.
[51] K.L. McMillan, Interpolation and SAT-based model checking, in: W.A. Hunt Jr., F. Somenzi (Eds.), Proc. of the 15th Conference on Computer-Aided Verification, CAV, in: LNCS, vol. 2725, Springer, July 2003, pp. 1–13.
[52] K.L. McMillan, N. Amla, Automatic abstraction without counterexamples, in: H. Garavel, J. Hatcliff (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, TACAS, in: LNCS, vol. 2619, Springer, April 2003, pp. 2–17.
[53] A. Miné, A new numerical abstract domain based on difference-bound matrices, in: PADO II, in: LNCS, vol. 2053, Springer–Verlag, May 2001, pp. 155–172.
[54] A Miné, The octagon abstract domain, in: AST 2001 in WCRE 2001, IEEE, IEEE CS Press, October 2001, pp. 310–319.
[55] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Design Automation Conference, 2001.
[56] M.R. Prasad, A. Biere, A. Gupta, A survey of recent advances in SAT-based formal verification, Software Tools for Technology Transfer (STTT) 7 (2) (2005) 156–173.
[57] Robby, M.B. Dwyer, J. Hatcliff, Bogor: An extensible and highly-modular software model checking framework, in: 9th European Software Engineering Conference, ACM Press, 2003, pp. 267–276.
[58] R. Rugina, M.C. Rinard, Symbolic bounds analysis of pointers, array indices, and accessed memory regions, in: Conference on Programming Language Design and Implementation, ACM Press, 2000, pp. 182–195.
[59] S. Sankaranarayanan, M. Colón, H.B. Sipma, Z. Manna, Efficient strongly relational polyhedral analysis, in: VMCAI, in: LNCS, Springer-Verlag, 2006.
[60] S. Sankaranarayanan, F. Ivančić, A. Gupta, Program analysis using symbolic ranges, in: 14th International Static Analysis Symposium, SAS, in: LNCS, vol. 4634, Springer, August 2007, pp. 366–383.
[61] S. Sankaranarayanan, F. Ivančić, I. Shlyahkter, A. Gupta, Static analysis in disjunctive numerical domains, in: K. Yi (Ed.), 13th International Static Analysis Symposium, SAS, in: LNCS, vol. 4134, Springer, August 2006, pp. 3–17.
[62] L. Séméria, G. deMicheli, SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C, in: International Conference on Computer-Aided Design, IEEE/ACM, November 1998, pp. 321–326.
[63] M. Sheeran, S. Singh, G. Stalmarck, Checking safety properties using induction and a SAT solver, in: Conference on Formal Methods in Computer-Aided Design, in: LNCS, vol. 1954, Springer, November 2000, pp. 108–125.
[64] W. Simpson, PPP: The point-to-point protocol, RFC 1661, IETF, June 1994.
[65] SystemC, http://www.systemc.org.
[66] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (1995) 121–189.
[67] W. Visser, K. Havelund, G. Brat, S. Park, Model checking programs, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, 2000.

[68] D. Wagner, J. Foster, E. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, in: Proc. Network and Distributed Systems Security Conference, ACM Press, 2000, pp. 3–17.
[69] C. Wang, Z. Yang, F. Ivančić, A. Gupta, Disjunctive image computation for embedded software verification, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE, European Design and Automation Association, Leuven, Belgium, 2006, pp. 1205–1210.
[70] Z. Yang, C. Wang, A. Gupta, F. Ivančić, Mixed symbolic representations for model checking software programs, in: 4th ACM & IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE, IEEE, 2006, pp. 17–26.
[71] A. Zaks, S. Cadambi, I. Shlyakhter, F. Ivančić, Z. Yang, M.K. Ganai, A. Gupta, P. Ashar, Range analysis for software verification, in: International Workshop on Software Verification and Validation, SVV, August 2006.
[72] L. Zhang, S. Malik, Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications, in: Conference on Design Automation and Test in Europe, 2003.