# Model Checking C Programs Using F-SOFT

Franjo Ivančić*, Ilya Shlyakhter*, Aarti Gupta*, Malay K. Ganai*, Vineet Kahlon*, Chao Wang*, Zijiang Yang†
*NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540
†Dept. of Computer Science, Western Michigan University, Kalamazoo, MI 49008

***Abstract*— With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. This paper provides a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. We provide illustrative details of a verification platform called F-SOFT, which provides a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.**

## I. 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 [1]. 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 II.

While model checking of hardware designs and protocols has been extensively studied, its application to software verification had been limited to 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 [2]. The general approach is to extract suitable verification models from the given source code 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 (in C), 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 III.

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 IV.

In terms of general abstraction techniques, *predicate abstraction* has emerged to be a popular technique for extracting verification models from software [3], [4], [5], [6]. Details of predicate abstraction and refinement, along with recent improvements, are described in Section V. Basically, 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 can contain 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* [7], [8] demonstrated for hardware designs.

We have developed a prototype software model checking tool called F-SOFT [9], which utilizes many of the ideas presented here. This is described in detail in Section VI. F-

Proceedings of the 2005 International Conference on Computer Design (ICCD'05)
0-7695-2451-6/05 $20.00 © 2005 IEEE

SOFT 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 checkers for a set of standard programming bugs such as array bound violations, NULL pointer dereferences, use of uninitialized variables, memory leaks, 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 perfomed by a tool called DiVer [10], which includes several state-of-the-art symbolic model checking techniques.

The outline of the paper is as follows. We start by providing a brief background on model checking in Section II. In Section III we discuss software modeling techniques that are useful for deriving verification models from C programs. We describe various back-end model checking techniques in Section IV, with an emphasis on heuristics targeted to improve verification efficiency on models generated from software programs. In Section V we discuss automatic predicate abstraction and refinement techniques. We present our prototype software verification tool F-SOFT in Section VI, along with detailed description of two verification case studies. Finally, we offer some concluding remarks in Section VII.

## II. BACKGROUND: MODEL CHECKING

This section provides a brief overview and terminology for model checking – more details can be found in the related book [1]. 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 error 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 [11], parameterized system verification [12], and software program verification [2], [13]. Furthermore, model checking techniques have also been extended to pushdown systems [14], [15], 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 III. These ideas also apply to extraction of pushdown models, such as Boolean programs [15].

Explicit state model checkers, such as SPIN [16], 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 [17], 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) [18], 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) [19], 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 [19] or proofs by induction [20], [21], 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 [22], [23], verification techniques based on SAT have become very popular (see a recent survey [24] 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 [25], [9]. A related important development has been the use of *resolution-based proof-analysis* techniques [26], [27] 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 [28], [29], refinement [30], and for interpolant-based verification [31], [32]. These methods allow SAT-based BMC to be combined effectively with other techniques to provide complete verification mtehods. There has also been growing interest in the use of SAT for unbounded model checking [33], [34], [35]. However, these techniques are not as robust as SAT-based BMC techniques.

## III. 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 [36], [37].

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. This representation is then converted to a Boolean representation by allocating latches to each C variable and converting next-state C expressions in terms of C variables into Boolean expressions in terms of the latches.

Formally, the transformations produce a verification model of the program. The model consists of a control flow graph (CFG) $G = (V, E)$ with a non-empty set of basic blocks $V = \{B_1, \ldots, B_n\}$. Each edge $e = (B_i, B_j, c_{ij}) \in E$ represents a guarded transition between basic blocks. For a given $i$, the conditions $c_{ij}$ are mutually exclusive, i.e. program flow is deterministic. The set of assignments in a basic block $B_i$ are rewritten to a parallel form as described later. We often use the term location $l \in V$ interchangeably with a basic block.

Let $X$ denote the set of all variables in the program. 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 $\{e_1, \ldots, e_n\} \subseteq \Sigma$.

We define a *state* of a program to be a tuple $(l, \boldsymbol{x})$, consisting of a location $l \in V$ representing the basic block, and a type-consistent valuation of data variables $\boldsymbol{x} \in \mathcal{X}$, where out-of-scope variables at $l$ are assigned the undefined value $\perp$. We consider the initial state of the program to be an initial location $l_0$, where each variable in $X$ can take any value that is type-consistent with its specification. The set of *initial states* is thus $Q_0 = \{(l_0, \boldsymbol{x}) \mid \boldsymbol{x} \in \mathcal{X}\}$. For checking reachability in programs, we define a set of blocks Bad $\subseteq V$ 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 $(l_0, \boldsymbol{x_0}), \ldots, (l_{k-1}, \boldsymbol{x_{k-1}})$ such that $(l_0, \boldsymbol{x_0}) \in Q_0$ is an initial state and $\forall 0 \leq i < k-1 : (l_i, \boldsymbol{x_i}) \rightarrow (l_{i+1}, \boldsymbol{x_{i+1}})$, where $\rightarrow$ denotes a transition between the states. A counterexample of length $k$ is a path that ends in an unsafe location, that is $l_{k-1} \in$ Bad.

### A. 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 multiplexing expressions as described below.

**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 [38] 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,...,z` at a given program location, we can rewrite a pointer read `*(p+i)` as a conditional multiplexing expression of the form `((p+i)==&a ? a : ((p+i)==&b ? b : ...))` where `&a,&b,...` are the numeric memory addresses we assigned to the variables `a,b,...` respectively.

**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, along with code to save and restore local state for recursive functions only.

### B. Modeling C Control Flow

In this section, we discuss the modeling techniques employed to handle control flow constructs.

**Functions.** We make all variables global[1], and move the code of all functions into `main()`. Each function is inlined exactly once; function calls are replaced with `goto`s to the function's first statement. Parameters and return values are passed via global variables, by adding assignments at each function call. Function return is handled by storing a unique id of the call site in a global variable before the call, and replacing `return`s with groups of `goto`s conditioned on this variable. An alternative is to inline each function at each call site [25], but this can significantly increase the model size.

**Control flow graph.** The C program now consists of labeled blocks of assignments followed by conditional `goto`s, giving a control flow graph (CFG) illustrated in Figure 1 on a small example. We rewrite the assignments within each basic block, so that their parallel execution corresponds to the original sequential semantics: if a variable `v` is assigned expression `e` in some statement, all appearances of `v` on the right-hand side of subsequent statements are replaced with `e`. We can prune irrelevant blocks by *backward slicing* from the error block, i.e., by removing those blocks whose execution can not affect the program's ability to reach the error block. Finally, we add a variable `pc` representing the program counter, to encode the number with which we identify each basic block (shown as numbers inside hexagons in Figure 1).

We can now construct symbolic transition relations for `pc`, and for each data variable appearing in the program. For `pc`, the transition relation reflects the guarded transitions between basic blocks in the CFG. Formally, every edge $e = (B_i, B_j, c_{ij})$ contributes to the transition relation of the `pc`, such that $pc' = B_j$ iff $(pc == B_i) \wedge c_{ij}$, for all $i, j$, where `pc` (`pc'`) refers to the old (new) value of the program counter. For the example in Figure 1, we have `pc'= (pc==0? 1 : (pc==1? (x>4? 3 : 2):...))`.

For a data variable, the transition relation is built from expressions assigned to the variable in various blocks. Let $e_i$ denote the right hand side (rhs) of an assignment to $x$ in block $B_i$ if $x$ is assigned in $B_i$. Then $x' = e_i$ iff $(pc == B_i)$, for all $i$. For example, in Figure 1, the variable `t` is assigned in blocks 6 and 7 and its value is unchanged in all other blocks; we have `t' = (pc==6 ? t-3 : (pc==7 ? l+2 : t))`.

Finally, we construct a Boolean formula representation of these transition relations resembling a hardware circuit. For the `pc` variable, we allocate $\lceil \log N \rceil$ latches, where $N$ is the total number of basic blocks. For each C program variable, we allocate a vector of $n$ latches, where $n$ is the bitwidth of the variable. For example, for a variable of type `int`, we typically allocate 32 latches. We also translate next-state expressions written in terms of C variables into vectors of next-state Boolean functions written in terms of the associated latches. For example, to translate addition of two $n$-bit integer values we construct an $n$-bit binary adder; to translate a relational expression, we construct a binary comparator, etc. In order to reduce the number of latches and the associated logic for each C variable, we can use static analysis to bound the range of values of each variable (see Section VI-A). The correctness property for the C program "can error label be reached" translates to "can the latches representing the `pc` variable take the value of the error block number".

[1]This is for the purpose of explanation here; in actual practice the back-end model checker can quantify out local variables earlier.

```
int foo(int s){
  int t=s+2;
  if (t>6)
    t -= 3 ;
  else
    t--;
  return t;
}

void bar(){
  int x = 3,
  y = x - 3;
  while (x<=4){
    y++ ;
    x = foo(x);
  }
  y = foo(y);
}
```

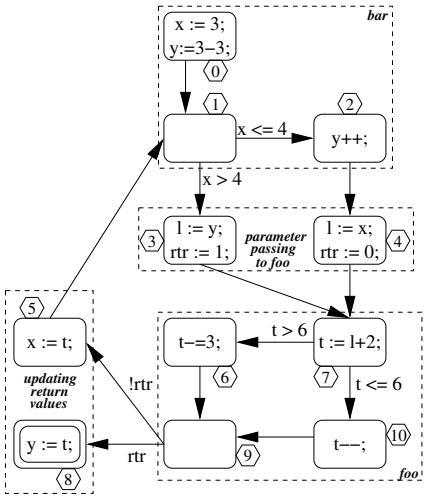


Fig. 1. Computing the control flow graph

## IV. Verifying Software Models

While conversion of C programs to a finite state representation allows application of well-studied model checking techniques, it can also lose much of the high-level information present in the programs. There are many ways to incorporate such high-level knowledge in back-end model checking techniques. The two most popular are: by use of additional high-level constraints on the model description, and by controlling model checker parameters such as decision heuristics in a SAT solver. As shown by experimental results reported in Section VI, these techniques can significantly impact the performance of back-end model checking.

### A. SAT-based Model Checking

Consider verification performed by SAT-based BMC, where each unrolling corresponds to a block-wise execution of the program. For verification models extracted from CFGs, as described in the previous section, the control flow is intuitively more important than the data flow. This is because each control state (the program counter) uniquely determines which data variables get updated, and which control states the program can possibly go to in the next state (depending on the transition guards). A good SAT heuristic is to increase the decision score of program counter variables relative to the data variables. This allows the SAT solver to make decisions first on control flow, rather than on data flow. In particular, an assignment to the program counter variables at a given time step, immediately makes constraints arising from other basic blocks irrelevant at

that time step. Thus, even though an unrolling for SAT-based BMC may include a copy of the transition relations for all basic blocks, most of them quickly become irrelevant once the SAT solver chooses a particular program path to explore.

The importance of control flow can be further emphasized by adding a "one-hot encoding" of the program counter in the verification model. In this encoding, a new binary-valued variable is introduced for each basic block in the CFG, such that the variable is true if and only if the program control is in that block. Correspondingly, in the BMC unrolling, each unrolled one-hot variable is true if and only if the program control is in that block at the associated time step. By introducing one-hot variables, the SAT solver can make "word-level" decisions on the program counter: with a single decision on a one-hot variable, the SAT solver effectively assigns all bits of the binary-encoded program counter. Again, the decision scores of these one-hot variables can be increased, making it likely that the SAT solver will assign these variables early in the search.

Another kind of high-level information that can be exploited concerns the transition structure of the CFG. For example, each basic block typically has a small number of predecessors. Additional constraints can be added to ensure that the choice of an active block $b$ in a given time frame $k$ restricts the choice of the active block in time frame $k-1$ to be one of the immediate predecessors of $b$ in the CFG. Note that these constraints are already implied by the transition relation for the *pc* variable, i.e. these constraints are redundant. However, adding them explicitly can increase the efficiency of the SAT search. On the other hand, adding too many redundant constraints can slow down the SAT solver, and negate the advantage of explicitly restricting the search space.

### B. BDD-based Model Checking

The above description focused on use of SAT-based BMC. However, the finite state models extracted from C programs, with or without use of predicate abstraction, allow use of unbounded verification methods also. In particular, standard BDD-based reachability computation can be used to compute the set of reachable states, and to prove the absence of errors.

BDDs are very sensitive to the number of state variables. It is therefore preferable to keep the model sizes small, e.g. by not using the one-hot encoding described above. Furthermore, additional high-level information can be used to reduce the number of state variables. Examples of such techniques include use of range analysis for statically bounding the values of data variables (Section VI-A); and use of register sharing techniques to reduce the number of control variables (Section V-B). Other improvements include use of disjunctively-partitioned image computation [39], which works better for software models than the conjunctively-partitioned technique used typically for hardware designs.

## V. Predicate Abstraction and Refinement

Model checking suffers from the state explosion problem, and *abstraction* is an important technique for reducing the state explosion problem [1]. In the domain of software verification, predicate abstraction has emerged to be a popular technique for extracting verification models from source code [6]. It abstracts data by keeping track of certain predicates on the data, but keeps the control flow of the original program. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The original program, also called the concrete model, is thus abstracted using a set of predicates into an abstract model, as shown in Fig. 2.

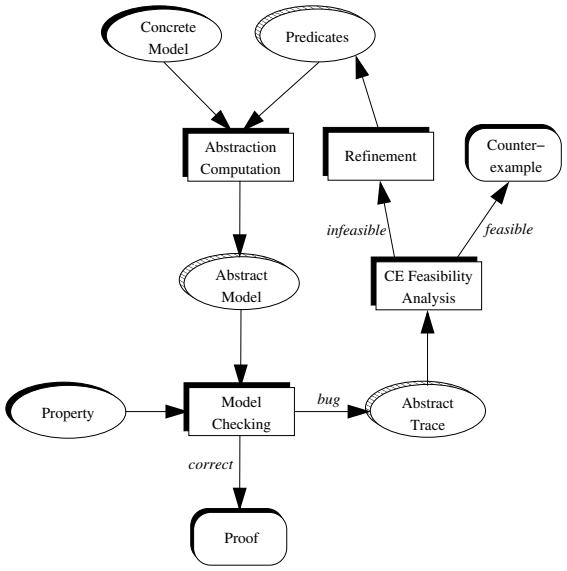


Fig. 2. Predicate abstraction refinement loop

The abstract model and the (reachability) property is then passed on to a model checker. If the property is proved correct on the abstract model, then the property is also valid on the concrete model. However, the abstract model may contain so-called *spurious* counterexamples that do not correspond to any feasible counterexample in the original system. Such spurious counterexamples are prevented by producing a more detailed abstract model using a *refinement* of the abstraction. This process is iterated as shown in Fig. 2 until the property is either proved or disproved.

### A. Abstraction Computation

Given a concrete software model represented as a control flow graph $G = (V, E)$, and an $n$-dimensional tuple of predicates $\boldsymbol{P} = (P_1, \ldots, P_n)$ over program variables, an abstract state $q$ is defined as a pair consisting of a location $l$ and a combination of Boolean values $\boldsymbol{b}$, i.e. $q = (l, \boldsymbol{b}) \in Q^{\boldsymbol{P}} = V \times \mathbb{B}^n$, where $\mathbb{B} = \{0, 1\}$. The set of initial abstract states is $Q_0^P = \{(l_0, \boldsymbol{b}) | \boldsymbol{b} \in \mathbb{B}^n\}$. An abstract state usually corresponds to a set of concrete states which can be computed using the concretization function $\gamma^{\boldsymbol{P}} : \mathbb{B}^n \rightarrow 2^{\mathcal{X}}$ defined on the truth value vector $\boldsymbol{b}$ as:

$$\gamma^{\boldsymbol{P}}(\boldsymbol{b}) = \{\boldsymbol{x} \in \mathcal{X} | b_1 \Leftrightarrow P_1(\boldsymbol{x}) \wedge \ldots \wedge b_n \Leftrightarrow P_n(\boldsymbol{x})\}.$$

For each basic block, the abstraction tracks how the truth values of the Boolean predicates are affected by statements in the block, given the truth values of the Boolean predicates when entering the block. If we consider the concrete transition relation $\rightarrow \subseteq Q \times Q$ as a representation of the steps in the CFG of the original program, we can then define the abstract transition relation $\rightarrow^{\boldsymbol{P}} \subseteq Q^{\boldsymbol{P}} \times Q^{\boldsymbol{P}}$ given a vector of predicates $\boldsymbol{P} = (P_1, \ldots, P_n)$ as:

$$(l, \boldsymbol{b}) \rightarrow^{\boldsymbol{P}} (l', \boldsymbol{b'}) \quad :\Leftrightarrow \quad \exists \boldsymbol{x} \in \gamma^{\boldsymbol{P}}(\boldsymbol{b}), \boldsymbol{x'} \in \gamma^{\boldsymbol{P}}(\boldsymbol{b'}). \; (l, \boldsymbol{x}) \rightarrow (l', \boldsymbol{x'}).$$

In other words, there exists a transition between two abstract states if and only if there exists a transition in the concrete model between some pair of corresponding concrete states.

We can then obtain the abstract transition relation $\mathcal{T}(\boldsymbol{b}, \boldsymbol{b'})$ for a basic block with assignments of expressions $e_1, \ldots, e_k$ to the set of program variables $Y$ as:

$$\mathcal{T}(\boldsymbol{b}, \boldsymbol{b'}) \quad := \quad \exists \boldsymbol{x} \in \mathcal{X}. \bigwedge_i b_i \Leftrightarrow P_i(\boldsymbol{x}) \wedge \bigwedge_i b'_i \Leftrightarrow P_i[Y \hookleftarrow e_1, \ldots, e_n](\boldsymbol{x}).$$

Here, $P_i[Y \hookleftarrow e_1, e_2, \ldots, e_n]$ denotes a point-wise substitution in $P_i$ of the $Y$ variables by the corresponding expressions in the assignments.

The abstract transition relation can be computed using decision procedures such as theorem provers or SAT-solvers [34]. However, building the most accurate abstract model is often prohibitively expensive, and may not be required for verification purposes. In order to reduce the abstraction computation time, various heuristics have been proposed to compute *approximate* or *coarse* abstract models. In Microsoft's SLAM [40], for example, coarse abstractions are generated using techniques such as *Cartesian approximation* and the *maximum cube length approximation* [41]. These techniques limit the number of predicates in each decision procedure call.

As an illustrative example, consider the program given on the left hand side of Figure 3. The property to be analzyed is whether the label `ERROR` is reachable in the program. As can be easily seen, the label is actually not reachable. Fig. 3 also shows an abstraction of the source code by using two predicates `b1` and `b2`. Here, these predicates appear in the two conditions in the program, namely `x==m`, represented by `b1`, and `y!=m+1` represented by `b2`. Note that for illustration purposes, we show the abstraction on a statement-by-statement basis, instead of showing it for basic blocks. We use the symbol `*` to represent a nondeterministic choice; that is, in line (1), the predicate `b1` can nondeterministically take the value `true` or `false`.

Since the assignment to variable `x` in line (1) does not affect either `y` or `m`, it does not impact the predicate `b2`. This assignment does change the value of `x`, which means that the predicate `b1` may be impacted. In this case however, since no relationship between `c` and `m` is known or representable using the current set of predicates, the assignment to `x` causes a

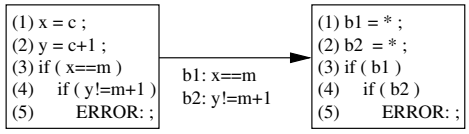


Fig. 3. Predicate abstraction computation using two predicates `b1` and `b2`

nondeterministic update to `b1`. Similar reasoning can be used to understand the translation of the remaining statements. In particular, note that the two `if`-conditions in lines (3) and (4) are translated to simple checks on `b1` and `b2`, respectively, as shown in the abstract model on the right in Fig. 3.

Recent efforts [32], [42] describe improved approaches, where predicates can be added locally to certain basic blocks, but not to others, which we will call henceforth *localization of predicates*. On average the number of predicates at each program location is small and thus, localization of predicates enables the abstraction computation to scale to larger software programs. Details about the refinement techniques employed are discussed in Sec. V-D.

### *B. Model Checking the Abstract Model*

The generated abstract model can also be represented as a CFG where all data variables are of Boolean type. The previously described approaches for verifying the resulting models (described in Section IV can therefore be directly applied. In particular, the BDD-based model checkers are often used in practice, to try and complete successful proofs of correctness, since this guarantees correctness on the original program as well. The performance of BDD-based model checkers depends crucially on the number of state variables. When the number of predicates is large, model checking of the abstract model can become a bottleneck even with a symbolic representation of the state space. In [42], the locality of predicates is used to reduce the number of (Boolean) state variables in the abstract model. The fact that each predicate is only locally useful can be used to represent different predicates in different parts of the program by same state variable. We call the reuse of state variables in the abstract model *register sharing*.

Register sharing thus enables more efficient model checking of the abstract models. However, maximal register sharing might also result in a large number of refinement iterations as described in the following. Consider a sequence $SE$ of statements from $s$ to $s'$, which does not modify the value of a predicate $P$. Suppose $P$ is localized at the statements $s$ and $s'$, but not at any intermediate statement in $SE$. In abstraction with register sharing, $P$ may be represented by two different Boolean variables $b_1$ and $b_2$ at $s$ and $s'$, respectively. Because the value of $P$ remains unchanged along $SE$, the value of $b_1$ at $s$ should be equal to the value of $b_2$ at $s'$. If this is not tracked, we may obtain a spurious counterexample by assigning different values to $b_1$ at $s$ and $b_2$ at $s'$. This problem can be avoided, if $P$ is represented by one Boolean variable $b$ in a large scope of the abstraction. We call a Boolean variable which represents only one predicate for a

large scope a *dedicated state variable*. The decision about when to assign a dedicated Boolean variable to a predicate can be made heuristically, based on the the fraction of blocks where a predicate is used.

Returning back to our example in Fig. 3, the model checker may find an abstract counterexample that first sets `b1` to `true` in line (1), then sets `b2` to `true` in line (2), and then proceeds through the `if`-statements to reach the `ERROR` label. The next step is to analyze this abstract counterexample to check whether it is feasible in the concrete model or not.

### C. Counterexample Feasibility Analysis

In order to check if a sequence of statements in the `C` program is (in)feasible, one can use either a theorem prover or a SAT-solver. Formally, we define an *abstract path of length* $k$ to be a sequence of $k$ abstract states $(l_0, \boldsymbol{b_0}), \ldots (l_{k-1}, \boldsymbol{b_{k-1}})$ such that $(l_0, \boldsymbol{b_0}) \in Q_0^{\boldsymbol{P}}$ is an initial abstract state and $\forall 0 \leq i < k-1 : (l_i, \boldsymbol{b_i}) \rightarrow^{\boldsymbol{P}} (l_{i+1}, \boldsymbol{b_{i+1}})$, where $\rightarrow^{\boldsymbol{P}}$ denotes a transition in the abstract model. An *abstract counterexample of length* $k$ is a path that ends in an unsafe location, that is $l_{k-1} \in$ Bad.

For the counterexample feasibility analysis, we define an expression $\varphi$ in the concrete state space that corresponds to a prefix of the counterexample of length $k$. We define prefixes for $0 \leq i < k$ as: $\varphi(i) := T^0 \wedge \ldots \wedge T^i$, where $T^i$ denotes the unrolled concrete transition relation at step $i$. In case that $\varphi(k-1)$ is satisfiable, we have found a counterexample in the original program, and the verification is complete. If the expression $\varphi(i)$ is unsatisfiable for some $0 \leq i < k$, we have discovered that the counterexample is indeed spurious. In this case, we need to perform a refinement of the abstraction as discussed in the following.

Reconsider the example presented in Fig. 3 and the abstract counterexample discussed in Section V-B. The expression generated for the counterexample is `x=c` $\wedge$ `y=c+1` $\wedge$ `x=m` $\wedge$ `y`$\neq$`m+1`. It is easy to see that this expression is unsatisfiable, thereby showing that this trace is indeed spurious.

### D. Refinement

There are two possible reasons for a spurious counterexample in the abstract model. The first reason is that the considered predicates are not adequate to prove correctness of the property. One possible approach to discover new predicates is based on *weakest pre-conditions* at the point of infeasibility of the abstract trace. For example, if $\varphi(i)$ is satisfiable, but $\varphi(i+1)$ is not, then using the weakest pre-conditions of changed predicates in basic block $l_i$ (or its predecessor blocks using the assignments in those blocks) can provide a new set of predicates that will eliminate this spurious counterexample [**?**]

Consider again the example in Fig. 3. While the expression `x=c` $\wedge$ `y=c+1` $\wedge$ `x=m` is satisfiable, it can be seen that adding the conjunct `y`$\neq$`m+1` makes the expression unsatisfiable. By performing a backward search of the abstract trace to find the last assignment to either `y` or `m` before the last `if`-statement, we can find the assignment to `y` at line (2). Computing the weakest pre-condition for the predicate `y!=m+1` for this assignment `y=c+1` provides a new predicate `c!=m`. In the refinement step, this new predicate is added to the abstract model (as predicate `b3`). The abstraction computation is performed again with the updated set of predicates, with the resulting abstract model as shown in Figure 4. Since predicates `b1` and `b2` hold different values in the abstract model, the label `ERROR` cannot be reached anymore, thereby proving the orignal program correct.

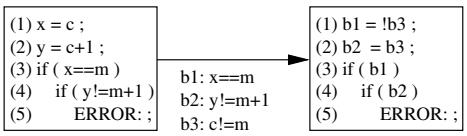


Fig. 4. Refinement step

The second reason for a spurious counterexample is that a coarse abstraction might have been computed, which omitted certain details about the relationship between predicates in the abstract model. While the first reason requires additional predicates, the second reason can be resolved by adding more precision to the abstract model based on the same predicates. In the SLAM toolkit, for example, such spurious behavior based on inexact predicate relationships is removed by a separate refinement algorithm called CONSTRAIN [43].

The BLAST tool [44] introduced the notion of *lazy abstraction*, where the abstraction refinement is completely demand-driven to remove spurious behaviors. In [32], a new refinement scheme based on interpolation [45] is described, which exploits the unsatisfiable core generated from a proof of unsatisfiability, to add new predicates to some program locations only. Our contribution in F-SOFT [42] is inspired by the lazy abstraction approach and the localization techniques implemented in BLAST [32]. Given an infeasible trace, and the unsatisfiable core from the proof of unsatisfiability, we find predicates whose values need to be tracked at each statement in order to eliminate the infeasible trace. For any program location we only need to track the relationship between the predicates relevant at that location. Furthermore, since we use predicates based on weakest pre-conditions along infeasible traces, most of the predicate relationships are obtained from the refinement process itself. This enables us to significantly reduce the number of calls to back-end decision procedures leading to a faster abstraction computation.

To illustrate the localized predicate abstraction approach with register sharing, consider Figure 5. As explained in [42], the set of predicates is computed using weakest pre-condition propagation. For this example, the same set of predicates as shown in Fig. 4 is computed. However, the localization approach limits the scope of each predicate. For example, predicate `x==m` is used only after statement (1) until statement (3). Similarly, the predicate `y!=m+1` is used only after statement (2) until statement (4). Lastly, the predicate `c!=m` is needed only before statement (2) is executed. We can thus define a mapping from the three predicates (`b1, b2, b3`) to two registers (`r1, r2`) depending on the program locations.

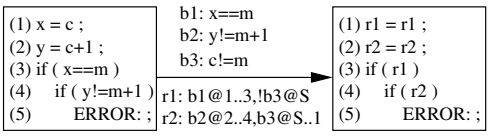


Fig. 5. Localized predicate abstraction with register sharing

As shown in the figure, `r1` represents first the negation of predicate `b3` at the start (`S`) of the program, while it represents predicate `b1` after the first statement is executed. Similarly, `r2` represents initially the predicate `b3`, and the predicate `b2` after statement (2) is executed. Since `r1` and `r2` represent `!b3` and `b3`, respectively, we add an additional constraint to the model `r1`$\neq$`r2`. The abstraction in Figure 5 is thus enough to prove the correctness of the program.

## VI. F-SOFT TOOL OVERVIEW

In this section we describe our prototype model checking tool F-SOFT [9], shown in Figure 6. In the front-end, we first use CIL [46] to make all expressions side-effect-free (adding temporary variables as needed), to make all identifiers globally unique, and to rewrite complex `C` constructs in terms of simpler ones (e.g. `switch` and `for` in terms of `if` and `goto`). Next, we perform various static analyses, such as computing the control flow graph of the program, performing program slicing with respect to the property and performing range analysis as described later in this section. Then, software modeling is performed to extract a finite state model represented as a circuit, as described earlier in Sec. III, along with information for additional heuristics for SAT-based BMC. Optionally, a localized predicate abstraction with register sharing can also be performed, as described in Sec. V. The back-end verification of the resulting verification models is performed by the DIVER [10] tool, which includes several BDD-based and SAT-based model checking techniques. If a true counterexample is discovered, a testbench program (in `C`) is automatically generated, which can be executed in the user's favorite debugger for analyzing the trace.

### A. Range Analysis

Since model checking suffers from the state explosion problem, which is further exacerbated in the context of software verification, it is important to reduce the sizes of the abstract models where possible. In F-SOFT we efficiently determine conservative ranges for values of program variables by performing static analysis. Such range information for each variable can be used to generate smaller models, both in terms of number of state elements and in the size of the datapath logic. This improves the efficiency of back-end model checking engines, especially those based on BDDs. Furthermore, additional range constraints can also help in improving the efficiency of predicate abstraction, by constraining the search space of the concrete model while computing the abstraction. Although range analysis techniques have been used for other applications [47], [48], [49], [50], we believe we are the first to use them for software model checking.

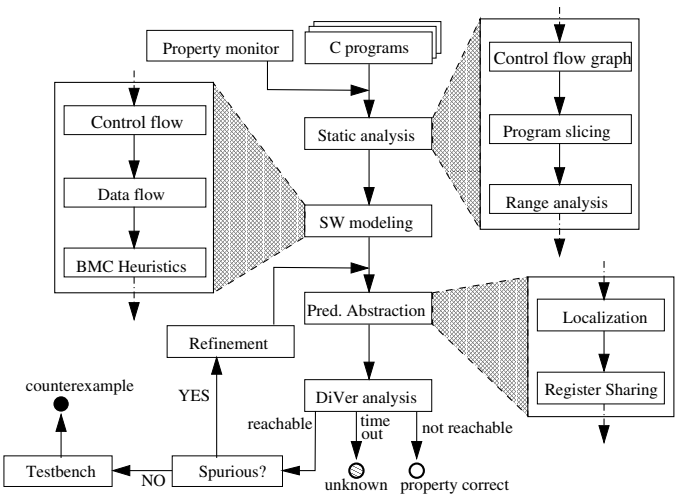


Fig. 6. F-SOFT tool overview

Our main method is based on the framework suggested in [49] which formulates each range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to an LP (linear programming) problem, which can be analyzed by any available LP solver. The solution to the LP problem provides symbolic lower and upper bounds for the values of all variables. More details on the range analysis computation used in F-SOFT and the heuristics targeted for its application to software verification are discussed in [51].

### B. Testbench Generation and Debugging

As mentioned earlier, the back-end verification is performed using the DIVER tool. If DIVER 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` in the lower half of Figure 7. In this example, F-SOFT performed the check for use of uninitialized variables. Here, F-SOFT reports that a bug was `found in file test.c at line 48 (block #23): 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. Figure 7 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

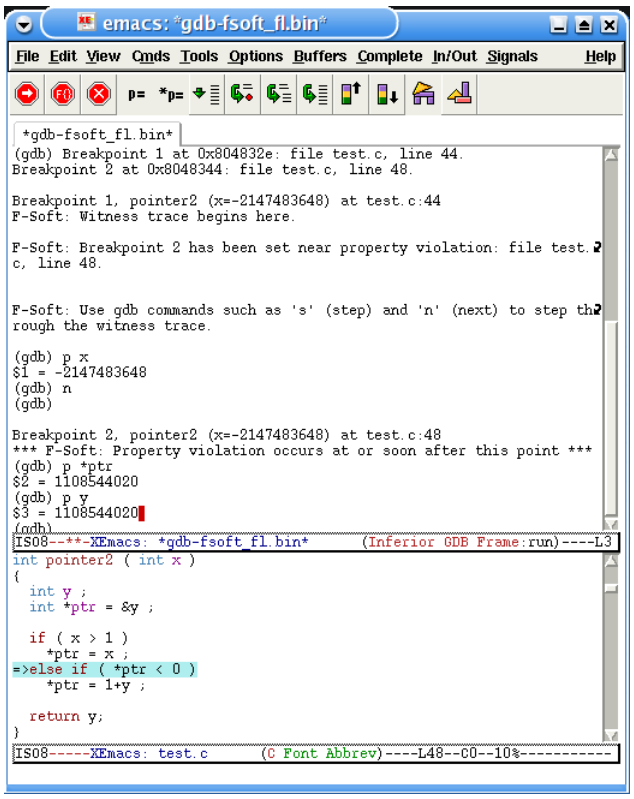


Fig. 7. F-SOFT testbench generator for gdb in emacs

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 Figure 7 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 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.

### C. Verification Case Studies

In this section, we describe two verification case studies for F-SOFT. The first is a network protocol called the Point-to-Point Protocol (PPP), which we used to evaluate our software modeling techniques and various heuristics for SAT-based BMC. The second is a TCAS (Traffic Alert and Collision Avoidance System) case study, for which we used the predicate abstraction framework in F-SOFT(More details on these case studies can be found in related publications [52], [42].)

**PPP Case Study**: We followed a previous attempt [53] to verify a part of the PPP protocol with respect to its specification defined in a Request for Comment (RFC) document. RFC 1661 [54] specifies the state transition table of an automaton with 10 states, which reacts to 15 events. The automaton can switch states when receiving an event, and also perform other actions, such as sending replies. Any implementation of

| | 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 |

TABLE I
A PART OF THE PPP SPECIFICATION

the PPP has to follow this behavior described in the RFC, which is partly shown in Table I 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 considered 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;
    }
}
```

We wanted to verify that the public implementation adheres to the specification as given in RFC 1661. In [53], the C-program as described here, was manually translated to the input language of the model checker MOCHA [55]. Their analysis showed that the public implementation does not fully adhere to the specification given by RFC 1661. In particular, when a peer receives a packet RTA, it is supposed to send back a configuration request, which is implemented correctly. However, it is also supposed to update its internal state to Req-Sent, which is missing in the implementation. In contrast to the MOCHA approach, we performed model checking directly on the source code (after slight modifications).

We first employed F-SOFT's range analysis framework (described in Section VI-A) to reduce the number of bits required to model the source code. After range analysis, the model contains 258 state bits, in contrast to 1435 state bits without use of range analysis. Even on this reduced model, the BDD-based model checker was not able to complete verification within a 3-hour time limit. On the other hand, SAT-based BMC was able to find the bug at an unrolling

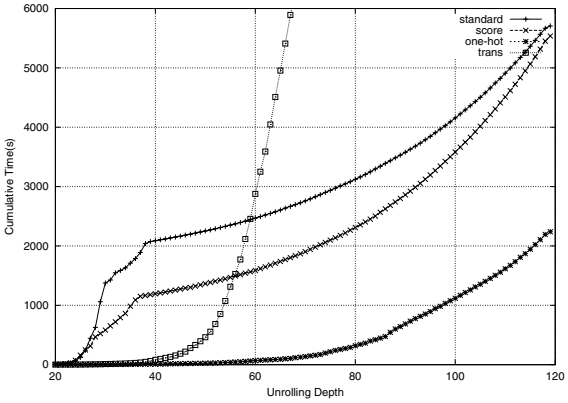


Fig. 8. Cumulative time comparison of BMC heuristics for the PPP example

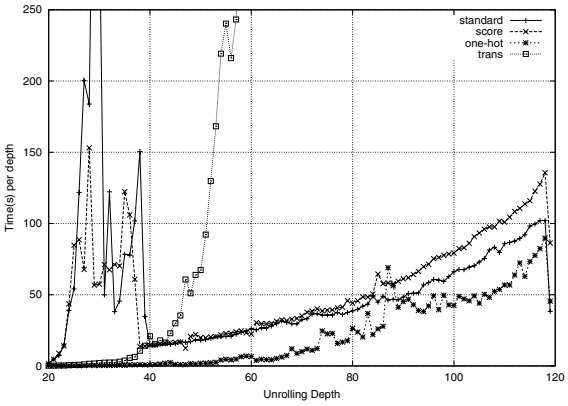


Fig. 9. Time per depth comparison of BMC heuristics for the PPP example

depth of 119, in about 95 minutes. The addition of one-hot encoding for the program counter added about 900 bits into the model. However, this heuristic allowed the SAT-solver to discover the bug in about 37 minutes. Additional introduction of the explicit CFG predecessor constraints did not improve the verification performance much (36 minutes). On the other hand, adding only these predecessor constraints, without the use of one-hot encoding of the program counter, overburdened the SAT solver, causing it to time-out in 3 hours.

Figures 8 and 9 show a more detailed comparison of the individual heuristics and their performance for the PPP case study. Fig. 8 shows the cumulative time in seconds taken for all depths up to a given number, while Fig. 9 shows the respective time needed for a particular depth. The graphs labeled *standard* represent the standard decision heuristics implemented in the DIVER tool. The graph shows the advantage of using 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 SAT-based BMC using the heuristics of higher decision scores for pc variables (*score*), one-hot encoding (*one-hot*) and addition of CFG predecessor constraints (*trans*). It is noteworthy that the inherent learning in the SAT-solver is preserved, which is visible by the various peaks in the respective graphs. Fig. 8, in particular, shows the advantage of the one-hot encoding heuristic for the PPP case study which consistently outperforms the other heuristics.

**TCAS Case Study**: We used an ANSI-C version of a TCAS component available from Georgia Tech. Even though the pre-processed program has only 1652 lines of code, the number of predicates needed to verify the properties is non-trivial for both F-SOFT and BLAST [32]. We checked 10 different safety properties of the TCAS system, and the results are shown in Table II. Each property was encoded as a certain error label in the code. If the label is not reachable, then the property is said to hold. Otherwise, we report the length of the counterexample in the "Bug" column. CPU times are given in seconds, and we set a time limit of one hour for each analysis.

We first experimented with no localization of predicates. However, this approach did not scale, as the abstraction computation becomes a bottleneck. We next experimented with localization of predicates using weakest pre-conditions. The results of applying only localization and abstraction *without* register sharing is shown under the "Localize" heading in the table. The "Time Abs MC" column gives the total time, followed by the breakup of total time into the time taken by abstraction (Abs), model checking (MC), respectively. We omit the time taken by refinement, which is equal to Time - (Abs + MC) for each row. The "P" and the "I" columns give the total number of predicates, and the total number of iterations, respectively.

Two observations can be made from the "Localize" results: 1) Due to the localization of predicates, the abstraction computation is no longer a bottleneck. 2) Model checking takes most of the time, since for each predicate a state variable is created in the abstract model. The model checking step is the cause of the timeouts in three rows under the "Localize" results.

Next, we experimented with register sharing. The number of state variables in the abstraction was reduced, and the individual model checking steps became faster. However, this approach resulted in too many abstraction refinement iterations. This problem was solved by discovering on-the-fly whether a predicate should be assigned a dedicated state variable, that is, a state variable which will not be reused. In these experiments, a dedicated state variable is allocated for a predicate whose usage exceeds a progressively increasing threshold, starting at 5% of the total number of program locations.

The results of combining these multiple techniques is given under the "Combined" heading in Table II. The "P Max Ded" column gives the total number of predicates (P), followed by the maximum number of predicates active at any program location (Max), and the total number of state variables which represent exactly one predicate, that is, dedicated state variables (Ded). Observe that the time spent during model

| Bench -mark | Localize | | | | | Combined | | | | | | | Bug |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Abs | MC | P | I | Time | Abs | MC | P | Max | Ded | I | |
| TCAS0 | 245 | 7 | 196 | 71 | 32 | 36 | 5 | 15 | 65 | 26 | 18 | 31 | - |
| TCAS1 | 1187 | 15 | 1069 | 108 | 44 | 161 | 9 | 118 | 96 | 35 | 25 | 38 | - |
| TCAS2 | 952 | 10 | 882 | 74 | 38 | 104 | 25 | 51 | 95 | 31 | 24 | 36 | - |
| TCAS3 | 940 | 15 | 864 | 91 | 36 | 46 | 17 | 17 | 73 | 22 | 15 | 33 | 152 |
| TCAS4 | 1231 | 13 | 1111 | 97 | 39 | 88 | 9 | 48 | 90 | 34 | 25 | 32 | 166 |
| TCAS5 | 1222 | 11 | 1128 | 79 | 41 | 141 | 8 | 98 | 98 | 37 | 29 | 31 | - |
| TCAS6 | TO | 20 | 2270 | 117 | 49 | 330 | 16 | 266 | 109 | 40 | 33 | 40 | 179 |
| TCAS7 | 1758 | 16 | 1627 | 79 | 47 | 64 | 10 | 29 | 94 | 28 | 21 | 33 | 160 |
| TCAS8 | TO | 21 | 1988 | 84 | 51 | 119 | 13 | 68 | 106 | 34 | 27 | 41 | - |
| TCAS9 | TO | 26 | 3349 | 113 | 58 | 250 | 14 | 186 | 106 | 34 | 27 | 44 | 179 |

TABLE II

RESULTS FOR: 1) LOCALIZATION, ABSTRACTION WITHOUT REGISTER SHARING ("LOCALIZE") . 2) LOCALIZATION, ABSTRACTION WITH REGISTER SHARING, DEDICATED STATE VARIABLES ("COMBINED"). A "-" INDICATES THAT THE PROPERTY HOLDS. A "TO" INDICATES A TIMEOUT OF 1HR.

checking (MC) has reduced significantly as compared to the "Localize" column.

### D. Comparison with Related Tools

In terms of the use of SAT-based BMC for software verification, the most closely related work to F-SOFT is the CBMC tool [25]. CBMC translates a C program into a Boolean formula, by considering bounded unrollings of loops, and uses a back-end SAT solver to find reachable error states. However, there are many differences. One major difference is that F-SOFT generates a finite state model (not just a formula) from the C program. This model can be analyzed by both SAT-based (bounded and unbounded) and BDD-based model checking techniques. Another major difference in the software modeling is the block-based approach used in F-SOFT rather than a statement-based approach in CBMC. (In our controlled experiments, the block-based approach provides a typical 25% performance improvement over a statement-based approach.) Additionally, the translation to a Boolean formula in CBMC requires unwinding of loops up to some bound, a full inlining of functions, and it cannot handle recursive functions. In contrast, the translation method in F-SOFT does not require unwinding of loops, avoids multiple inlinings, and can also handle bounded recursion. This allows F-SOFT to scale better than CBMC on larger programs, especially those with loops.

We also differentiate the F-SOFT approach by use of light-weight pre-processing analyses such as *program slicing* and *range analysis*. Program slicing is used to statically remove parts of the given program that do not affect the property of interest. It has been successfully used in many software model checkers [2], [13], although most of these are explicit state model checkers. We believe F-SOFT is the first to use static range analysis to reduce the size of the extracted model for the purpose of model checking. This provides considerable savings in comparison to a full bitwidth encoding, as in CBMC. Finally, F-SOFT also allows abstraction of the software program using predicate abstraction and localization techniques. Indeed, the use of this framework in F-SOFT has been inspired by other efforts, including SLAM [40] and BLAST [32].

## VII. CONCLUSIONS

This paper provided a brief tutorial on model checking of C programs. The essential approach is to model the semantics of C programs in the form of finite state systems by using suitable abstractions. The use of abstractions is key, both for modeling programs as finite state systems and for reducing the model sizes in order to manage verification complexity. This paper provided illustrative details of a verification platform called F-SOFT, which entails a range of abstractions for modeling software, and uses customized SAT-based and BDD-based model checking techniques targeted for software.

**Acknowledgements.** We thank Pranav Ashar, Srihari Cadambi, Himanshu Jain, and Aleksandr Zaks for their help in the F-SOFT effort.

## REFERENCES

[1] E. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2000.

[2] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[3] P. Cousot and 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.

[4] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, ser. LNCS, vol. 1254. Springer, 1997, pp. 72–83.

[5] S. Das, D. Dill, and S. Park, "Experience with predicate abstraction," in *Computer Aided Verification*, ser. LNCS 1633. Springer, 1999, pp. 160–171.

[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani, "Automatic predicate abstraction of C programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 203–213.

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

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, 2000, pp. 154–169.

[9] F. Ivančić, Z. Yang, I. Shlyakhter, M. Ganai, A. Gupta, and P. Ashar, "F-SOFT: Software verification platform," in *Computer-Aided Verification*, ser. LNCS. Springer, 2005.

[10] M. Ganai, A. Gupta, and 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*, ser. LNCS, vol. 3340. Springer, 2005.

[11] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993.
[12] M. Browne, E. M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Information and Computation*, vol. 81, pp. 13–31, 1989.
[13] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from Java source code," in *International Conference on Software Engineering*, 2000, pp. 439–448.
[14] A. Boujjani, J. Esparza, and O. Maler, "Reachability analysis of push-down automata: Applications to model checking," in *CONCUR'97: Concurrency Theory, Eighth International Conference*, ser. LNCS 1243. Springer, 1997, pp. 135–150.
[15] T. Ball and S. Rajamani, "Bebop: A symbolic model checker for Boolean programs," in *SPIN 2000 Workshop on Model Checking of Software*, ser. LNCS 1885. Springer, 2000, pp. 113–130.
[16] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
[17] K. McMillan, *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
[18] R. Bryant, "Graph-based algorithms for boolean-function manipulation," *IEEE Transactions on Computers*, vol. C, no. 35, pp. 677–691, 1986.
[19] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and 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.
[20] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT solver," in *Conference on Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 1954. Springer, November 2000, pp. 108–125.
[21] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs Complement SAT-based BMC in DiVer," in *Proc. of the* $15^{th}$ *Conference on Computer-Aided Verification (CAV)*, ser. LNCS, W. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, July 2003, pp. 206–209.
[22] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.
[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference*, 2001.
[24] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Software Tools for Technology Transfer (STTT)*, vol. 7, no. 2, pp. 156 –173, April 2005.
[25] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.
[26] L. Zhang and 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.
[27] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Conference on Design Automation and Test in Europe*, 2003.
[28] K. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, April 2003, pp. 2–17.
[29] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *Intern. Conf. on Computer Aided Design*, November 2003, pp. 416 – 423.
[30] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis," in *Proc. of the* $4^{th}$ *Conference on Formal Methods in Computer-Aided Design (FMCAD)*, ser. LNCS, M. Aagaard and J. O'Leary, Eds., vol. 2517, November 2002, pp. 33–51.
[31] K. McMillan, "Interpolation and SAT-based Model Checking," in *Proc. of the* $15^{th}$ *Conference on Computer-Aided Verification (CAV)*, ser. LNCS, J. W.A. Hunt and F. Somenzi, Eds., vol. 2725. Springer, July 2003, pp. 1–13.
[32] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan, "Abstractions from proofs," in *Symposium on Principles of Programming Languages*. ACM Press, 2004, pp. 232–244.
[33] A. Gupta, Z. Yang, P. Ashar, and 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*, ser. LNCS 1954. Springer, 2000, pp. 354–371.
[34] K. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Computer Aided Verification*, ser. LNCS, vol. 2404. Springer, 2002.
[35] M. Ganai, A. Gupta, and P. Ashar, "Efficient memory modeling for SAT-based BMC," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, 2004.
[36] G. d. M. L. Séméria, "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.
[37] SystemC, "http://www.systemc.org."
[38] 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.
[39] S. Barner and I. Rabinovitz, "Efficient symbolic model checking of software using partial disjunctive partitioning," in *The 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*. Springer-Verlag, Oct. 2003, lNCS 2860.
[40] T. Ball and S. Rajamani, "The SLAM toolkit," in *Computer Aided Verification, 13th International Conference*, 2001.
[41] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and Cartesian abstraction for model checking C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2031. Springer-Verlag, April 2001.
[42] H. Jain, F. Ivančić, A. Gupta, and M. Ganai, "Localization and register sharing for predicate abstraction," in *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 3340. Springer, 2005, pp. 397–412.
[43] T. Ball, B. Cook, S. Das, and S. Rajamani, "Refining approximations in software predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 388–403.
[44] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Symposium on Principles of Programming Languages*, 2002, pp. 58–70.
[45] W. Craig, "Linear reasoning," in *Journal of Symbolic Logic*, 1957, pp. 22:250–268.
[46] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. LNCS. Springer-Verlag, 2002, pp. 213–228.
[47] W. Blume and R. Eigenmann, "Symbolic range propagation," in *Proceedings of the 9th International Symposium on Parallel Processing*, 1995, pp. 357 – 363.
[48] P. Cousot and N. Halbwachs, "Automatic discovery of linear constraints among variables of a program," in *Proc. of the 5th ACM Symp. on principles of programming languages (POPL)*, 1978, pp. 84–96.
[49] R. Rugina and M. 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.
[50] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *PLDI*, 2000, pp. 108 – 120.
[51] A. Zaks, I. Shlyakhter, F. Ivančić, H. Cadambi, Z. Yang, M. Ganai, A. Gupta, and P. Ashar, "Range analysis for software verification," 2005, in submission.
[52] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," in *Symposium on Leveraging Formal Methods in Applications*, 2004.
[53] R. Alur and B. Wang, "Verifying network protocol implementations by symbolic refinement checking," in *13th Conference on Computer Aided Verification*, 2001, pp. 169–181.
[54] W. Simpson, "PPP: The Point-to-Point Protocol," IETF, RFC 1661, June 1994.
[55] R. Alur, L. de Alfaro, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Kirsch, and B. Wang, "MOCHA: A model checking tool that exploits design structure," in *Intern. Conf. on Software Engineering*, 2001, pp. 835–836.