Modeling, Abstraction and Analysis of Software using Boolean Techniques

Franjo Ivančić, Malay Ganai, Aarti Gupta, Pranav Ashar
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540

Zijiang Yang
(Consultant to NEC Laboratories)
Western Michigan University
Dept. of Computer Science
Parkview Campus
Kalamazoo, MI 49008

ABSTRACT
This paper discusses the formal analysis and automatic verification of software programs using our prototype tool FSOFT. It is currently applicable to a subset of the C programming language allowing bounded recursion. We consider reachability properties, in particular whether certain assertions or basic blocks are reachable in the source code. We perform this analysis via a translation to a Boolean representation based on a modeling framework founded upon basic blocks. The program is then analyzed by a back-end SAT-based bounded model checker. Each unrolling is mapped to one step in a block-wise execution of the program. This paper is the first to use such efficient block-based unrollings for bounded model-checking. In addition, inspired by the current work on predicate abstraction we allow a partial abstraction of the software using an efficient SAT-based computation of the program elements to be abstracted. Additionally, we propose to analyze the feasibility of fragments of counter-examples in order to discover stronger reasons and thus more efficient predicates for refinement purposes in a counter-example-guided refinement approach. We also present a network protocol case study that is focused on our software modeling framework, and show experimental results for various heuristics used in the back-end SAT-solver that help improve the verification performance.

Keywords
Software Verification, Software Modeling, Bounded Model Checking, Satisfiability Solver, Predicate Abstraction

Categories and Subject Descriptors
D.2.4 [Software]: Software Engineering—Software/Program Verification; F.3.1 [Theory of Computation]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSSTA 2004 Boston, MA
Copyright 2004 ACM X-XXXX-XX-X/XX/XX...
$5.00.

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 [12]. In particular, model checking is automatic, and, if the design contains an error, model checking produces a counter-example (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. Traditionally, model checking algorithms use transition relations between states that are represented explicitly, and thus enumerate all reachable states explicitly. In [27], the use of binary decision diagrams (BDDs) was introduced to provide a canonical and compact form for Boolean formulas as a symbolic representation for state transition graphs. As the symbolic representation captures some of the regularity in the state-space, it enables the verification of systems with large number of states.

While symbolic model checking algorithms using BDDs offer the potential of exhaustive coverage of large state-spaces, it often still does not scale well enough in practice. An alternative approach is bounded model checking or BMC [7] focusing on the search for counter-examples of bounded length only. Effectively, the problem is translated to a Boolean formula, such that the formula is satisfiable if and only if there exists a counter-example of length $k$. In practice, $k$ can be increased incrementally starting from one to find a shortest counter-example if one exists. However, additional reasoning is needed to ensure completeness of the verification when no counter-example exists [24, 34]. The satisfiability check in the BMC approach is typically performed by a back-end SAT-solver. Due to the many advances in SAT-solving techniques [19, 26, 29], BMC can often handle much larger designs than BDDs.

Software modeling. In this paper, we describe our approach and tool development for the automatic analysis of software programs. Our initial prototype tool is developed for the C programming language. Similar to work described in [13], we translate a program into a Boolean representation to be analyzed by a back-end SAT-based BMC. We currently consider reachability properties, in particular whether certain (labeled) blocks are reachable in the source code. We perform this analysis via a translation to a circuit repre-
sentation by considering the control and data flow of the program. The control logic of the resulting circuit describes the flow of control that can be represented by a control flow graph. The data logic describes the assignment of variables given finite ranges. In contrast to [13], our procedure is based on translating blocks instead of individual statements as atomic components of programs.

The salient feature of our approach for software verification is the central role a basic block plays – from the modeling of software, the abstraction of the source code, to the verification or model checking steps of the generated software model using an unrolling based on blocks – the block modeling approach is integral to the proposed method. For each basic block in the source code we generate a label. Assuming the source code consists of $N$ basic blocks, we represent each basic block by a label consisting of $\lceil \log N \rceil$ bits. A program counter variable consisting of $\lceil \log N \rceil$ bits is introduced to monitor progress in the graph consisting of basic blocks.

The key contribution of this paper is to use this program counter variable to track progress of the allowed executions of the source code during the model checking phase, where the SAT-based BMC is performed by understanding an unrolling during the BMC to be one step in a block-wise execution of the program. In this context each atomic step of the BMC consists of an unrolled basic block rather than individual statements.

**Abstraction techniques.** Abstraction is probably the most important technique for reducing the state explosion problem in model checking [12]. In the world of program analysis, predicate abstraction has emerged to be a powerful and popular technique for extracting finite-state models from complex, potentially infinite state systems [5, 16, 17, 20]. Inspired by the work on predicate abstraction, we augment our software modeling approach with such an abstraction technique by considering a set of $n$ Boolean predicates represented by C expressions (without Boolean connectives).

In contrast to most other approaches, we are interested in updating the truth values of these predicates across basic blocks. We thus abstract the code of the blocks by their transition relational equivalents in terms of the current-state to next-state predicate variables. It should be noted that in this combined framework it is also possible to abstract certain elements of the code while leaving other parts (such as certain variables) in their concrete representation.

**Symbolic predicate abstraction.** In contrast to initial work on predicate abstraction (such as [5, 22]), which uses extensive and inherently expensive calls to theorem provers to compute the transition relations in the abstract system, we follow a more efficient SAT-based approach. A similar approach has recently been advocated and shown advantageous over the use of theorem provers [25]. Concurrent to our work, [14] describes a similar theoretical approach of using a SAT-based computation of the transition relation for the analysis of C-programs which has not been implemented yet.

In contrast to [14, 25], our implementation of the enumeration of the various transition relations reuses certain common computations thus allowing faster overall computation. Additionally, since many transition relations need to be computed where the set of considered Boolean variables often remains partially the same, an intelligent scheduling of the enumeration of these transition relations and a proper memory management of derived implications can potentially reduce the computation time of the overall abstraction procedure significantly.

As pointed out in [15] however, building the most refined abstract model even in a SAT-based approach is often too expensive and also not needed. Instead, in order to reduce the overall run-time by reducing the abstraction time, we allow the computation of coarsely abstracted models. Allowing an over-approximation of the considered abstraction model thus reduces the computation time of the abstraction, but also increases the risk that spurious counter-examples will appear during the model checking phase.

In [15] the notion of *spurious transitions* was introduced to deal with the fact that certain transitions in the coarsely abstracted model are by themselves not feasible in the concrete system, and an algorithm was proposed for handling such spurious transitions. In addition to a special treatment of spurious transitions, we generalize the concept of spurious counter-examples to test for the feasibility of sub-paths or fragments.

**Tool overview.** Figure 1 summarizes the overall approach as it is described in this paper. The input to F-Soft is a set of files corresponding to a software module represented as a C program in the figure. Furthermore, the input also consists of the property to be checked which is specified by a property monitor. The flow-chart recapitulates the above description of the software verification tool F-Soft, including the static analyses, abstraction techniques, and counter-example-guided refinement procedure.

Our boolean verification framework called DiVer [21] uses various SAT-based and BDD-based methods for performing both bounded and unbounded verification. We are able to adjust and include new decision heuristics in DiVer.
which will be able to take advantage of the fact that we are considering a Boolean design automatically abstracted and generated from software. One such simple decision heuristics, namely increasing the likelihood that the SAT-solver decides on variables first that correspond to the control flow of the program rather than the data flow is described in section 3.2. It was shown very successful in our experiments which are presented in section 4.2. In addition, DiVer provides us the advantage to use both bounded and unbounded model checking in the same framework, while, to the best of our knowledge, all other current software analysis approaches either use BDDs (such as SLAM [6]) or a SAT-based approach (such as [13]).

If the DiVer analysis proves the property not reachable in the (abstracted) Boolean model, it is guaranteed that the property is not reachable in the original software source code ending the analysis procedure in the white dot in figure 1. However, if DiVer finds a counter-example in the (abstracted) Boolean model, and the analysis of this counter-example finds that the path is actually feasible in the original source code, the tool has discovered a concrete counter-example in the original system ending the analysis procedure in the black dot in the figure.

Overview. In the next section we introduce some basic definitions used throughout this paper, while section 3 discusses our block-based software modeling approach. In section 4 we present a network case study that is analyzed using the tool F-Soft. Section 5 then discusses our efficient implementation of the symbolic predicate abstraction method, while section 6 discusses briefly our counter-example analysis and refinement approach. It should be noted that we only have initial indications of the effectiveness of the predicate abstraction and refinement approach, and are still in the process of completing the implementation. Section 7 concludes this paper with some remarks and an overview of future work.

2. BASIC DEFINITIONS

From now on we consider a program to consist of a non-empty set of blocks. We perform various automatic code simplification steps that do not affect the semantics of the program. These simplifications for example include transformations of expressions with side-effects into separate statements. The simplifications also include handling of repeat assignments to the same variable in the same basic block. Prior assignments to variables used for other assignments later in the same basic block are also resolved by substitution. Each block then contains a set of parallel assignments followed by a guarded choice of transitions leading to other blocks.

Assume that the set of all variables in the program is denoted by $X$. We denote a type-consistent evaluation of all variables in $X$ by $\mathbf{x}$, and the set of all type-consistent evaluations is denoted by $X$. While the set of allowed $\Sigma$-expressions is represented by $\Sigma$. Then, the parallel assignments of a block can be written as $\mathbf{v}_1, \ldots, \mathbf{v}_n \leftarrow \mathbf{e}_1, \ldots, \mathbf{e}_n$, where $V = \{v_1, \ldots, v_n\} \subseteq X$ and $E = \{e_1, \ldots, e_n\} \subseteq \Sigma$. We call $V$ the assignment set of a block. We also denote the set of variables that are in scope in a block $l$ as $X_l$ and the set of type-consistent evaluations with $X$. We use a function $\operatorname{Vars}: \Sigma \rightarrow 2^X$ to denote the set of variables (and their pointers and dereferences) that occur in a $\Sigma$-expression $\sigma \in \Sigma$ using $\operatorname{Vars}(\sigma)$. We generalize this function naturally to $\operatorname{Vars}: 2^X \rightarrow 2^X$ as $\operatorname{Vars}(E) = \bigcup_{e \in E} \operatorname{Vars}(e)$. For a particular $\mathcal{C}$-block with assignment set $V \subseteq X$ we can then define the set of required variables $R$ as $R = \operatorname{Vars}(E)$ and the set of unused variables $U$ as $U = X \setminus (\operatorname{Vars}(V) \cup H)$. Formally, we define a state of a program to consist of a location $l \in L$ describing the current basic block and a type-consistent evaluation of data variables $\mathbf{x} \in X_l$ where out-of-scope variables at location $l$ are assigned the undefined value $\perp$. We are considering the initial state of the program to be completely random in a single location $l_0$, that is each variable in $X$ can take a value that is type-consistent with its specification. The set of initial states thus is $Q_0 = \{(l_0, \mathbf{x}) | \mathbf{x} \in X_{l_0}\}$. We include a (bounded) fixed-length static variable modeling a function call stack to the set of global variables, thus allowing modeling of bounded recursion.

As mentioned before, we currently focus on reachability in programs. We thus define a set of locations $\text{Bad} \subseteq L$ to be unsafe, and our model checking analysis tries to prove or disprove whether 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 $(l_0, \mathbf{x}_0), \ldots, (l_{k-1}, \mathbf{x}_{k-1})$ such that $(l_0, \mathbf{x}_0) \in Q_0$ is an initial state and $0 \leq i < k-1$: $(l_i, \mathbf{x}_i) \rightarrow (l_{i+1}, \mathbf{x}_{i+1})$, where $\rightarrow \subseteq Q \times Q$ follows a standard definition of the transition relation in the state space. A counterexample of length $k$ then is a path that ends in an unsafe location, that is $l_{k-1} \in \text{Bad}$.

3. MODELING AND ANALYSIS OF SOFTWARE

In this section, we describe our software modeling approach that is centered around the concept of a basic block. We perform reachability analysis via a translation to a circuit representation by considering the control flow and data flow of the program. The control logic of the translated circuit describes the flow of control that can be represented by a control flow graph. The data logic describes the assignment of variables given finite ranges.

For each basic block in the source code we generate a label. Assuming the source code consists of $N$ blocks, we represent each block by a label consisting of $\lceil \log N \rceil$ bits. A program counter variable is introduced to monitor progress in the graph consisting of these basic blocks. The control logic defines the transition relation for this program counter given the block graph and the conditions guarding some of the transitions between the blocks.

The key innovative contribution of this paper is to use this program counter to track progress of allowed executions of the code during the bounded model checking phase. The model checking analysis is performed by understanding an unrolling to be one step in a block-wise execution of the program where each atomic step consists of a basic block rather than an individual statement. To the best of our knowledge, this paper is the first to use such block-based unrollings for bounded model-checking.

We also differentiate our approach through the use of pre-processing analyses such as program slicing and range analysis as part of the tool framework. A slice of a program with respect to a set of program elements is a projection of the program that includes only parts that might somehow affect the considered set of program elements [11]. Furthermore, we use range analysis techniques [8, 9, 30, 33] to limit the
number of bits needed to represent various statements in the code compared to using (at least) 32 bits for each statement as implemented in [13]. In this context each block of the code is replaced by a tight bit-blasted version of the data variables as well as the control logic given the aforementioned program counter and the block labels.

In the following we first discuss some details of our software modeling approach which considers the control flow and data flow separately. Following that, in section 3.2 we then discuss our analysis approach in more detail.

3.1 Software Modeling Details

Our software modeling method is based on a translation of the program code to a Boolean representation by considering the control flow and data flow of the program. This section discusses some details of our translation for these two components.

3.1.1 Control Logic

The CFG of a program is a finite graph $G = (L, E)$ where $L$ consists of basic blocks (also called locations) and $E$ denotes the edges between blocks representing control transfer. Except for the first and last instructions, the instructions in each block have a unique predecessor and successor. A parameter entry_blk in the recursive function constructing the control flow graph is the entry block to the current statement. Depending on the type of the considered statement, it may or may not become part of entry_blk. In the following we present the modeling of some representative statements commonly found in programs:

- **if (cond) {body1} else {body2}**
  The condition cond joins the entry block because there is no branch between the previous statement and the condition. Since the control branches over cond, we initialize new blocks body1 and entry2 to be the entry blocks for the first statements for body1 and body2.
  The blocks of body1 and body2 are then created recursively. The CFG contains edges (entry_blk→entry1) with guard cond, and (entry_blk→entry2) with guard ¬cond.

- **while (cond) {body}**
  Unlike the condition in an if statement the condition in a while statement starts a new block cond_blk because it is the destination of multiple blocks. Since the control may transfer to body after cond, a new block body_blk is initialized and the blocks for body are created recursively. A new block exit_blk is initialized to be the block that contains the statement following the while statement. There are two guarded edges in the CFG, which are (cond_blk→body_blk) with guard cond, and (cond_blk→exit_blk) with guard ¬cond.

- **label:**
  A new block label_blk is initialized for the labeled statement. The reason to create a new block is that labeled statement may be the destination of goto statements. Note that no blocks contains more than one labeled statement.

- **goto label**
  If the statement labeled by label has already been parsed, an unconditional edge (entry_blk→label_blk) is created, where label_blk is the basic block that contains the labeled statement. Otherwise, the basic block that contains the goto statement is stored in a hash table which is used to create edges when a label statement is parsed.

- **function calls**
  After simplification of the program code, there are only two types of function calls, namely: foo(...) and var=foo(...). Additionally, code simplification removes nested or embedded function calls inside other function calls by adding temporary variables. An unconditional edge from the calling basic block to foo's first basic block is created. If the function call is of type var=foo(...), we add a statement assigning the return expression to var. The statement is added to a new block that is the exit block of the function call.

  For functions that are not called recursively in any manner, we add statements that assign actual parameters to the corresponding formal parameters if parameters are needed. For non-recursive functions the return point of the called function in the program is recorded as a special parameter. The returning transitions from the function call are guarded with checks on this special parameter. To allow modeling of bounded recursion we include a bounded function call stack for functions that can be called recursively. The function call stack is used to determine to which basic block to return once the computation is completed by adding a guard on the returning transition which checks the information stored on the function call stack.

- **default**
  By default a statement, such as an assignment, will not create any new basic blocks. In such a case we simply append the statement to the entry block.

If $N$ denotes the number of basic blocks in a C program, we can use $2\log N$ bits to express the control states. As shown in table 1, $c_1, c_2,\ldots, c_n$ denote the current state program counter variable, and $c'_1, c'_2,\ldots, c'_n$ denote the next state program counter where $n = \lceil \log N \rceil$. The $j$th table line represents a CFG edge $(v_i^{(1)}, v_i^{(2)},\ldots,v_i^{(n)} \rightarrow v_j^{(1)}, v_j^{(2)},\ldots,v_j^{(n)})$ with guard $k^j$, where $v_i^{(j)} \in \{0,1\}$ is an assignment to $c_i$ and $v_j^{(j)} \in \{0,1\}$ is an assignment to $c_j$. Based on the truth table, we are able to build the next state logic for each next state variable as:

$$c'_j = \bigvee_{v_j^{(j)}=1} \bigwedge_{p=1} c_p \wedge \bigwedge_{p=0} \neg c_p.$$ 

Figure 2 shows the computed control flow graph using our approach for the following simple C code. The example shows how the basic blocks are computed for various types of statement as discussed earlier. The source of the control flow graph is the top most node, while the sink of the graph is the highlighted and left-most node in the bottom row.

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$\cdots$</th>
<th>$c_n$</th>
<th>$c'_1$</th>
<th>$c'_2$</th>
<th>$\cdots$</th>
<th>$c'_n$</th>
<th>guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$\cdots$</td>
<td>$v_n$</td>
<td>$v'_1$</td>
<td>$v'_2$</td>
<td>$\cdots$</td>
<td>$v'_n$</td>
<td>$k^1$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\cdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\cdots$</td>
<td>$\vdots$</td>
<td>$k^m$</td>
</tr>
</tbody>
</table>

Table 1: Truth table for control logic
int foo(int l){
    int t=l+2;
    if (t>6)
        return t;
    else
        t--;
    return t;
}

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

The example shows in particular how non-recursive function calls are included into the control flow of the calling function. Note that a preprocessing analysis determines that in this particular example the 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.

The example also shows the code simplification that allows us to view the assignments inside each basic block to be parallel assignments without creating fresh variables for each statement. The source node rewrites the assignment to y which is y = x-3; given a prior assignment x = 3; to the statement y = 3-3; which can then be processed in parallel with the assignment x = 3; without conflict.

### 3.1.2 Data Logic

Next we consider how to create the data logic for the translated Boolean representation. All the variables after the simplification and the range analysis have finite domains. Assume we need t bits to represent a variable \( \text{var}_j \) with \( \text{var}_j(1 \leq j \leq t) \) being the current state bits and \( \text{var}_i'(1 \leq i \leq t) \) being the next state bits. Let \( \text{var} \) be assigned in blocks \( \{b_1, b_2, \ldots, b_k\} \) and not assigned in the remaining blocks \( \{b_{k+1}, \ldots, b_N\} \). The logic assigned to \( \text{var}_j \) is \( V_{b_i} \) at block \( b_i (1 \leq i \leq k) \). Also, let \( L_i \) be the index of the block \( b_i \). The data logic for \( \text{var} \) is \( \text{var}_j = (\bigvee_{i=1}^{k}(c_1 c_2 \cdots c_n = L_i) \land (V_{b_i})) \lor (\bigvee_{i=k+1}^{N}(c_1 c_2 \cdots c_n = L_i) \land (\text{var}_j')) \).

In order to obtain the logic for each variable assignment \( \text{var} = \text{expr} \), we build combinational circuits for \( \text{expr} \). In the following we present a few representative right-hand expressions commonly found in an assignment:

- **var'**
  
  In this case the variable \( \text{var} \) is assigned to the value of another variable \( \text{var}' \). If a circuit representation has not been built for \( \text{var}' \) yet, we create \( n \) bit latches where \( n \) is the bit-width of \( \text{var}' \). Otherwise, we simply connect the latches of \( \text{var}' \) to the current circuit.

- **num**
  
  In the case that variable \( \text{var} \) is assigned to a constant value \( \text{num} \), let \( b_1 \ldots b_t \) be the binary representation of \( \text{num} \). The circuit then is a vector out with \( \text{out}[i] = b_i \) for \( b_i \in 0, 1 \) and \( 1 \leq i \leq n \).

- **expr1 & expr2**
  
  The circuits are first built recursively for the two sub-expressions. Let vectors \( \text{vec1} \) and \( \text{vec2} \) be the outputs of the circuit representations for \( \text{expr1} \) and \( \text{expr2} \), and \( n \) be the bit-width of each vector. The circuit for \( \text{expr1} \) \& \( \text{expr2} \) contains \( n \) additional two-input AND-gates. The inputs to the \( i \)-th gate are \( \text{vec1}[i] \) and \( \text{vec2}[i] \) for \( 1 \leq i \leq n \), and the output becomes the output of the resulting circuit.

- **expr1 + expr2**
  
  Similarly, let vectors \( \text{vec1} \) and \( \text{vec2} \) be the outputs of the circuit representations for \( \text{expr1} \) and \( \text{expr2} \), and \( n \) be the bit-width of each vector. A full adder is created for each pair of corresponding \( \text{vec1} \) and \( \text{vec2} \) bits: \( \text{out}[i] = \text{vec1}[i] \oplus \text{vec2}[i] \oplus \text{carry in}[i] \) for \( 1 \leq i \leq n \). The logic for \( \text{carry in}[i] \) is \( (\text{vec1}[i] \land \text{vec2}[i]) \lor (\text{vec1}[i] \lnot \land (\text{carry in}[i] \lnot)) \lor (\text{vec2}[i] \land \text{carry in}[i] \lnot) \) for \( 2 \leq i \leq n \), and \( \text{carry in}[1] = 0 \). The vector \( \text{out} \) is the output of the resulting circuit.

- **expr1 == expr2**
  
  The circuit representation for a relational expression has one bit. Let vectors \( \text{vec1} \) and \( \text{vec2} \) be the outputs of the circuit representations for \( \text{expr1} \) and \( \text{expr2} \), and \( n \) be the bit-width of each vector. Additionally, \( n \) XNOR-gates are created: \( \text{o}[i] = \lnot(\text{vec1}[i] \oplus \text{vec2}[i]) \). Finally, an \( n \)-input AND-gate is created to generate the output of the resulting circuit: \( \text{out} = \land_{i=0}^{n-1} \text{o}[i] \).

Some special handling is required to generalize the above modeling to pointer variables. When a pointer variable is declared, additional variables are introduced. For example, the declaration \( \text{int ***p, ***q} \) creates four variables \( v_p''', v_p'', v_p', v_p \) for pointer \( p \), each of which has its own data logic, and analogously for \( q \). Implicitly \( v_p \) denotes the referenced value of \( v_p'' \), which in turn denotes the referenced value of \( v_p''' \) and so on. The generated variables are regular finite domain variables for which we use pointer-analysis techniques to determine which locations a pointer can point to. Note that an assignment to a pointer changes the face value as well as the referenced values, which results in additional assignments. If there is an assignment \( *p = *q \); in
the program, the following three assignments are generated: 
\( v_p = v_q, v_p = \overline{v_q}, v_p = v_q \). Similarly, \( p = q \); results in four assignments that include \( v_p = \overline{v_q} \).

A dereference in the C code also leads to additional variables and assignments. Let \( r \) be an integer variable. An assignment \( *p = q; \) first generates a new variable \( v' \) that denotes the dereferenced value of \( v_q \), and then creates two assignments \( v'_p = v'_q, v_p = v_q \). It should be noted that our implementation currently does not consider pointer arithmetic or arrays.

### 3.2 Model Analysis

As mentioned before, the key contribution of this paper is to use a program counter variable founded upon basic blocks to track progress of allowed executions of the code during the bounded model checking phase. The model checking analysis is performed by understanding an unrolling to be one step in a block-wise execution of the program. Therefore, this particular modeling and analysis approach is an efficient way of performing bounded model checking, since for each basic block there is only a limited number of possible successors. Given a single initial block label, there are thus only a limited number of possible next blocks reachable in new unrollings. Thus, albeit each unrolling introduces the whole code into the satisfiability problem, many blocks in the new unrolling can be declared unreachable by mainly considering the control flow of the software program.

As mentioned earlier, we are able to adjust the decision heuristics used by our DiVer verification engine to take advantage of the fact that we are considering a Boolean design generated from a piece of software. A simple decision heuristics 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 based on a static analysis of the control flow graph. We implement this heuristics 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. This heuristics is shown successful in experiments which are presented in section 4.2.

### 4. NETWORK CASE STUDY

The verification of various network protocols has been a popular application of model checkers [18]. 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 a RFC (Request for Comments) document. In addition to this work, there have recently been attempts to verify an implementation of a protocol with respect to its standard defined in a RFC [4]. In this section, we describe the Point-to-Point protocol, which is one of our network protocol case studies that we have analyzed using F-SOFT, in more detail. We focus our case studies initially on network protocols since similar applications represent the first future application area of this research inside our company.

#### 4.1 Point-to-Point Protocol

The Point-to-Point Protocol 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, and 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 [4], we focus on checking the implementation of the option negotiation automation of the LCP part of PPP for link establishment.

RFC 1661 [35] 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 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 which messages should be sent back if any and what the next state should be if there is a change of state. An empty field describes the fact that the automaton will simply ignore a received packet.

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopped</td>
<td>Term-Req</td>
<td>Req-Sent</td>
</tr>
<tr>
<td>Req-Sent</td>
<td>goto Stopping</td>
<td>Opened</td>
</tr>
<tr>
<td>RCA</td>
<td>goto Ack-Rcvd</td>
<td>RCA</td>
</tr>
<tr>
<td>RTA</td>
<td>goto Req-Sent</td>
<td>RTA</td>
</tr>
</tbody>
</table>

Table 2: A part of the PPP specification

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 it is given in RFC 1661. The current prototype implementation of our verification engine does not allow to verify multi-threaded code. We have thus implemented the following scheme to analyze the implementation of PPP which is illustrated in figure 3. To be able to check the public implementation we need a second peer to engage it. We have thus written 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 have also implemented a specification model of the RFC in C.\(^1\)

\(^1\) A second instance of the C-code representing the RFC specification is actually used as the environment peer. The envi-
The environment and the public implementation are engaged in communication, while the RFC specification “listens in” on messages received by the public implementation. Although the RFC specification tries to respond to these messages as well, its outgoing link is “grounded” and does not affect the environmental peer. Our analysis thus translates to the question whether the internal state of the public implementation always matches the internal state of the RFC-based specification.

In [4], the C-program as described here, has been manually translated to the input language of the model checker Mocha[3]. 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 [4] 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 is also supposed to update its internal state to Req-Sent, which is missing in the implementation.

4.2 Experimental Results

The PPP case study centers around a part of the public implementation available in many Linux distributions that describes the daemon. The size of the original source code considered in our case study is about 1400 lines of C code with an additional 600 lines corresponding to the prior discussed environment model in C. Simplification of the source code increases the size of the code to about 4000 lines.

The Boolean model corresponding to the PPP case study contains about 200 bit-variables and around 14k gates, with 30 primary input bits modeling non-deterministic choice. Our DiVer tool completed the BDD-based model checking process in just over 3 hours. It discovered around 7.5 \cdot 10^7 many reachable states where the longest path is around 1200 states deep.

Figure 4 shows the performance gains of a SAT-based BMC on one representative run for the PPP case study using our software modeling approach. The graph labeled Std represents the standard decision heuristics implemented in the back-end DiVer tool. The graph shows that the analysis using Std runs out of time fairly quickly. The other two graphs represent runs where we increased the likelihood that the back-end SAT-solver makes decisions first on variables that correspond to the control flow of the translated software just has the additional capability to initiate communication or to model external hardware failures.

5. SYMBOLIC PREDICATE ABSTRACTION

Abstraction is probably the most important technique for reducing the state explosion problem in model checking [12]. Inspired by the current work on predicate abstraction, a powerful technique for extracting finite-state models from complex systems, we develop a tool for the analysis of software on top of our internal verification framework and the aforementioned software modeling approach using an efficient SAT-based computation of the abstraction.

We consider a set \( P = \{p_1, \ldots, p_n\} \subseteq \Sigma \) of \( n \) Boolean predicates. We are interested in updating the truth values to these Boolean predicates after the execution of each block given the predicate values when entering the considered block. We define a function Prefs : \( 2^X \rightarrow 2^P \) that maps a set \( M \) of variables to the predicates that range over these variables and their references and dereferences

\[
Prefs(M) = \{p \in P | \text{Vars}(p) \cap M \neq \emptyset\}.
\]

For efficiency purposes, we try to limit the set of predicates that are used during the computation of the abstract transition relation. We thus define the set of required variables for a predicate \( p \) given a basic block with parallel as-
Although the variables in \( R(p) \) influence \( p \) directly, we need to include more predicates and variables into the analysis to reach the full precision provided by the predicate set \( P \). Consider, for example, the statement \( x = 3 \); and three predicates \( p_1 = x = 2 \), \( p_2 = x = 1 \), and \( p_3 = x = 0 \). To compute the most refined abstract transition relation we want to include the predicates \( p_j \) but also \( p_i \), into the analysis at this point. We thus define the set of required variables for a predicate \( p \) given a predicate set \( P \) as

\[
R(p, P) = R(p) \cup \{ x \in X | v \in R(p, P) \land q \in \text{Preds} \{ \{ v \} \} \},
\]

which can be computed iteratively. This definition can be generalized for a subset of predicates \( P \subseteq P \) to \( R(P, P) = \bigcup_{p \in P} R(p, P) \subseteq X \). The sets of unused variables for a predicate \( p \) for a set of predicates \( P \) can then be defined as \( U(p, P) = X \setminus \text{Preds}(R(p, P)) \). For a parallel assignment we define the set of predicates to be updated as \( P_U = \text{Preds}(V) \), and also the set of predicates required as \( P_R = \text{Preds}(P(P), R) \), while we define the set of unused predicates to be \( P_V = P \setminus (P_U \cup P_R) \).

For each predicate \( p \in P \) we add a new Boolean variable \( b \) to the abstraction of the program. For the set \( P = \{ p_1, \ldots, p_n \} \) we thus consider an abstraction based on the set \( B = \{ b_1, \ldots, b_n \} \). For notational convenience, we consider the evaluation of the Boolean representation often to be a vector \( b = (b_1, \ldots, b_n)^T \in B^n \) with \( B = \{ 0, 1 \} \) rather than a set.

In the following we discuss two ways of implementing the computation of a transition relation describing the current-state-to-next-state relationship for these Boolean variables \( b_i \). In our context, current-state of a Boolean variable denotes the evaluation of the Boolean variable when entering a new block, while next-state denotes the evaluation of the same variable after executing all statements in the block, that is just before exiting the block. It should be noted that for guarded transitions we compute predicate strengthenings based on a similar analysis, and that the initial state of the abstraction is completely random.

More formally, we define the set \( T = \{ 0, 1, \ldots, \gamma \} \supset B \) for abstraction purposes. The symbol \( \gamma \) corresponds to the same symbol in the concrete state-space, namely a predicate evaluates to \( \gamma \) when it is out-of-scope in a given block. The symbol \( * \) is used to denote the fact that although the predicate is defined in a given block we are not interested in its evaluation to true or false. We use this notation for the description of coarse abstractions. The set of abstract states \( Q^n \) for a vector of predicates \( P \in X^n \) of length \( n \) is thus \( Q^n = L \times T^n \), where \( L \) denotes the set of locations or basic blocks. Similar to the concrete state-space we define a location-specific set \( T^n_X \) which only contains consistent vectors \( b \) with respect to scope for a given location \( X \).

The concretization \( \gamma(b) \) of a Boolean predicate vector \( b = (b_1, \ldots, b_n)^T \in B^n \) is the set of true assignments to the expression \( \gamma(b) = \{ x \in X | b_i \leftrightarrow p_i(x) \wedge \ldots \wedge b_n \leftrightarrow p_n(x) \} \). Assuming we know the range of all variables in \( P_V \), we can generate a bit-blasted expression that can be used to enumerate all such assignments using the algorithm described in [28]. We use the range analysis framework to find an appropriate bit-width for each variable and statement in the program. Note also that it is easy to extend these definitions to the set \( \mathbb{B} \) as well as to extend it to abstract states. If we consider the concrete transition relation \(-\subseteq Q \times Q\), we can then define the abstract transition relation \(-\subseteq Q^n \times Q^n\) given a vector of predicates \( P \) as

\[
(\gamma(b_1), \gamma(b_2)) = \{ (b_1, b_2) \in T^n_X \times T^n_X | \gamma(b_1), \gamma(b_2) \in \gamma(b_1) \wedge (\gamma(b_1), \gamma(b_1)) \rightarrow (\gamma(b_1), \gamma(b_2)) \}.
\]

### 5.1 Computing a Transition Relation

If \( \sigma \in \Sigma \) is an expression, \( Y \subseteq X \) is a set of \( k \) variables, and \( \sigma_1, \ldots, \sigma_k \) are \( k \) expressions, then \( \sigma[Y = \sigma_1, \ldots, \sigma_k] \) denotes the expression obtained by point-wise substitution of the expressions \( \sigma_i \) for the variables \( Y \) in \( \sigma \). For a block with assignment set \( V \), a set of updated predicates \( P_V \), and a set of required predicates \( P_R \), we partition the set \( B \) accordingly into sets \( B_V \) and \( B_R \), corresponding to \( P_V \) and \( P_R \) respectively.

We can obtain an expression representing a transition relation \( T(b, b', X) \) for the next state of a Boolean variable \( b' \) for a Boolean variable \( b \in B \) on the current state Boolean variables in \( B^n \) as

\[
T(b, b', X) := \bigwedge_{b_i \in B} b_i \wedge p_i \wedge p_{i+1} \rightarrow \gamma(b_{i+1}).
\]

Similarly, we obtain an expression representing a transition relation \( T(b, b', X) \) for the next state of all Boolean variables \( b \in B_R \) depending on the current state Boolean variables in \( B_n \) as

\[
T(b, b', X) := \bigwedge_{b_i \in B} b_i \wedge p_i \rightarrow \gamma(b_{i+1}) \wedge \bigwedge_{b_i \in B} b_i \rightarrow \gamma(b_{i+1}).
\]

For notational convenience, we use \( B' \) to denote the set \( \{ b[0,j] \} \in B' \).

For each block we thus generate one transition relation \( T(b, b', X) \). Every satisfying assignment \( A \) of the transition relation represents a transition in the concrete system and its abstraction given the current set of predicates. In order to compute \( T(b, b') \) we project such a satisfying assignment onto the Boolean variables \( b \) and \( b' \), and thus obtain one possible transition in the abstracted system by defining an abstract satisfying assignment \( A_B \) as \( A_B = A \cup B \). Since we are only interested in finding all transitions in the abstract system, we use an efficient implementation of the well known enumeration algorithm described in [28]. The enumeration of abstract transitions proceeds by iterative addition of so-called blocking clauses that disable a previous solution to reappear. After adding the blocking clause \( \neg A_B \) to the Boolean expression \( T(b, b', X) \), the SAT-solver is then asked to find another satisfying assignment of the enriched problem formulation, if such an assignment exists.

### 5.2 Efficient Implementation

The implementation of the enumeration of the various transition relations is constrained to reuse certain common computations thus allowing faster overall computation. Since many transition relations need to be computed, where the set of considered Boolean variables often remains only partially the same, an intelligent scheduling of the enumeration of these transition relations and a memory management of \( ^2 \)Actually, the set \( B^n \) was computed on the basis of all predicates, and can be relaxed if a single predicate at a time is being considered.
derived implications, can potentially reduce the computation time of the overall abstraction procedure significantly.

It should be noted that in many cases the set of $n$ predicates does not imply that there are $2^n$ many consistent abstract interpretations of these predicates. In fact, as it turns out, often there are many redundant or parallel predicates where many combinations of these evaluations do not correspond to any concrete state. As discussed in [23] in the context of hybrid systems (mixed discrete-continuous systems [1]), a simple preprocessing step analyzing whether predicates are parallel to each other can save considerable runtime. Consider, for example, a case that involved 35 relevant predicates which could be divided into four sets of parallel predicates of sizes eight, eight, nine and nine and a single non-parallel predicate. Parallel predicates in our context includes predicates such as $2x - y^2 = \frac{21}{2}$, $2x - y^2 > 5 + \frac{5}{2}$ and $2x - 0.8z < y^2$. In a set of nine parallel predicates for example, there are at most ten consistent evaluations whereas a full search would loop over $2^9$ many combinations. In the previously mentioned case, the 35 predicates thus can at most represent $2 \cdot 9^2 \cdot 10^2 < 21^4$ many abstract consistent evaluations. This reduces the analysis (at least) by a factor of more than $2^{21} > 2 \cdot 10^6$.

In addition to the described analysis of parallel predicates, there are often other non-consistent evaluations of the predicates in the concrete state-space. For example, for a subset of separation predicates [32] which are predicates of the form $x_i - x_j \sim c$ with $\sim \in \{>, \geq\}$ and $c$ a constant, we can use the algorithm described in [36] to decide the valid combination for these predicates. Such reductions of the set of feasible valid combinations can be helpful for later processing steps if the set can be expressed concisely. A full enumeration of the feasible combinations may also be possible; however, the resulting set may not be concise enough. It is also helpful to discover certain relationships and implications during an enumeration and remember these for later computation steps.

For the enumeration of the transition relation from current-state to next-state Boolean variables, the effect of reducing the size of the feasible Boolean predicate truth combinations shows up both in the current-state representation as well as the next-state representation of these variables. In addition, an analysis of various computations and an intelligent scheduling of the enumeration of various transition relations can significantly save time during the computation of the abstraction. Consider, for example, the following code fragment which is often found in some similar form in code:

\[
\text{if (cond) } x++; \text{ else } x+=2;\]

For illustration purposes, we consider here two predicates $b_1$ representing $x > 2$ and $b_2$ representing $x > 3$. Then, the expression $x > 1$ represents the pre-condition for $b_1$ to be true in the then-part of the code fragment, written as $b_1^{(t)}$, while it also represents $b_2^{(e)}$. We can thus save one whole computation of transition relations by combining the two enumerations into one. We thus enumerate the following expression\(^3\)

\[
\begin{align*}
&b_1 \iff x > 2 \land b_2 \iff x > 3 \land b_1^{(t)} \iff x > 1 \land \\
&b_2^{(e)} \iff x > 2 \land b_1^{(e)} \iff x > 0 \land b_2^{(e)} = b_1^{(t)}.
\end{align*}
\]

Even if the computation cannot be combined in the way as presented above, it is often the case that the set of considered Boolean predicates and instructions is partially the same in various transition relations representing different blocks. An analysis of these cases and a scheduling that allows the enumeration of common parts first, can then be used to minimize the amount of total computations needed for all transition relations.

As described earlier, the enumeration of the transition relation is performed for each basic block of the program. However, it should also be noted that blocks can be broken up into multiple smaller ones if the transition relation cannot be enumerated efficiently for a large basic block. In the extreme case, we can keep splitting blocks until each atomic statement is contained in separate basic blocks, which is however neither desirable nor needed. It is clear though that this feature allows us to compute the transition relations even if a block contains too many assignments.

As pointed out in [15] however, building the most refined abstract model even in a SAT-based approach is often too expensive and also not needed. Instead, in order to reduce the overall run-time by reducing the abstraction time, we allow the computation of coarsely abstracted models. For example, while it may not be feasible to enumerate the set of possible solutions for the full transition relation between current-state and next-state Boolean predicate variables, it may be possible to reduce the run-time by enumerating multiple transition relations where only a subset or individual next-state Boolean predicate variables are considered at a time.

On the other hand, as suggested in [15], an approximation can also be computed based on the overall transition relation by excluding some abstracted satisfying assignments from further consideration. By limiting the number of significant predicates to a small constant, that is by iterating only over abstract states with a pre-determined number of 0’s or 1’s in their predicate vector representation and + and \_ appropriately elsewhere, we can simply construct an over-approximation of the transition relation using the algorithm described in [15].

A coarsely abstracted model can also appear due to lazy refinement; that is, the refinement procedure produces an enlarged set of predicates which is not updated fully on the whole program. It may also appear if our computation of $P_\eta$ that includes iterative computation of $R(p, \overline{P})$ is stopped prematurely before a fix-point is reached. Since we allow approximations of the transition relation $\sim \rightarrow^\eta$, we denote the approximate abstract transition relation as $\sim \rightarrow^\eta$ and perform the reachability analysis in general using the transition relation $\sim \rightarrow^\eta$.

6. COUNTER-EXAMPLE ANALYSIS AND REFINEMENT

Allowing an over-approximation of the considered abstraction model reduces the run-time of the computation of the abstraction, but also increases the risk that spurious counter-examples appear during the model checking phase. In [15] the notion of spurious transitions was introduced to deal with the fact that certain transitions in the coarsely abstracted model are by themselves not feasible in the concrete system, and an algorithm was proposed for handling such spurious transitions.

In addition to a special treatment of spurious transitions,
we generalize the concept of spurious counter-examples to test for the feasibility of sub-paths or fragments. The concept of spurious fragments has recently been used and shown advantageous for the verification of hybrid systems [10].

For hybrid systems, it has been proven that the analysis of spurious fragments yields stronger predicates for the following model checking iterations [2]. The additional work invested to discover better reasons for spurious counter-examples thus yielding stronger predicates, can reduce the number of iterations needed to prove or disprove reachability in the overall counter-example-guided abstraction refinement approach significantly.

The analysis of fragments will increasingly become more important as the counter-examples become longer. While the reason for a spurious counter-example may be a local problem in a sequence of relatively few abstract states, the prefix analysis as advocated in other approaches may distort the real reason and discover a sub-quality set of new predicates. A similar counter-example with a very similar spurious fragment can re-appear in the following iterations if it is not accounted for during prior refinement steps. We do not fully implemented the refinement procedure and predicate discovery approach yet.

For the counter-example analysis, we define an abstract path of length $k$ in the abstract state-space $Q^k$ for a set of predicates $P$ given a set of unsafe locations $\text{Bad} \subseteq L$ as a sequence of $k$ abstract states $(l_0, b_0), \ldots (l_{k-1}, b_{k-1})$ such that $(l_0, b_0) \in Q^0$ is an initial abstract state and $\forall 0 \leq i < k - 1 : (l_i, b_i) \rightarrow_P (l_{i+1}, b_{i+1})$. An abstract counter-example of length $k$ is a path that ends in an unsafe location, that is $l_{k-1} \in \text{Bad}$. However, since we allow approximate abstract transition relations we also define coarsely abstracted paths and coarsely abstracted counter-examples analogously using the relation $\preceq_P$ instead of $\rightarrow_P$.

For the counter-example analysis, we define an expression that corresponds to a fragment of the counter-example of length $k$. We define a timed version of the transition relation $\rightarrow$ in the concrete state-space for a time-step or unrolling $i$ with $0 \leq i < k - 1$ over the timed or unrolled variables as $T^i$. Since we know the sequence of blocks in a coarsely abstracted path $(l_0, b_0), \ldots, (l_{k-1}, b_{k-1})$ we can for efficiency purposes additionally limit the various timed transitions in the implementation to the appropriate transition in the control flow graph to the block transition from $l_i$ to $l_{i+1}$. We define fragments for $0 \leq i < j < k$ as:

$$\varphi(i, j) := \gamma(b_i) \wedge T^i \wedge T^j \wedge \gamma(b_{j+1}) \wedge \ldots \wedge T^{j-1} \wedge \gamma(b_j)$$

If the expression $\varphi(i, j)$ is unsatisfiable for any pair $0 \leq i < j < k$, we have discovered that the counter-example is indeed spurious.

The previously presented analysis and refinement algorithms consider the following two special cases: Spurious transitions as defined in [15] consider fragments of the form $j = i + 1$. Traditional prefix counter-example analysis algorithms consider only fragments of the form $i = 0$ with a minor addition. Namely, when including the first abstract state we also need to verify that a possible path through the counter-example starts within the set of initial states. However, in our case, since we define all well-defined states in $X_{0}$ in basic block $b_0$ to be initial, we can omit this check here. For the full counter-example analysis we simply consider the expression $\varphi := \varphi(0, k - 1)$. It should also be noted that for long counter-examples it may not be feasible to check all possible fragments which necessitates future research into heuristics for choosing fragments to be considered.

7. CONCLUSIONS AND FUTURE WORK

This paper introduces our software verification prototype tool F-Soft founded upon basic block software modeling and BMC-unrolling, and additionally includes a counter-example-guided predicate abstraction refinement routine. It is currently applicable for a subset of the C programming language allowing bounded recursion. We consider reactivity properties, in particular whether certain assertions or basic blocks are reachable in the source code. We translate a program into a Boolean representation to be analyzed by a back-end SAT-based BMC by understanding an unrolling during the BMC to be one step in a block-wise execution of the program. To the best of our knowledge, this paper is the first to use such block-based unrollings for bounded model-checking.

Our Boolean verification framework called DiVer [21] uses various SAT-based and BDD-based methods for performing both bounded and unbounded verification. We are able to adjust and include new decision heuristics in DiVer, which will be able to take advantage of the fact that we are considering a design automatically abstracted and generated from software. One such simple decision heuristics, namely increasing the likelihood that the SAT-solver decides on variables first that correspond to the control flow of the program rather than the data flow, was shown very successful in our experiments as presented in section 4. We are also applying our tool to other case studies, one of which is the analysis of another network protocol, namely the ad-hoc on-demand distance vector routing protocol (AODV) [31].

There remain many research directions to follow in the future. As already mentioned in the introduction, we are still in the process of completing the full implementation of the abstraction and refinement methods. Initial results are showing promising directions for future research.

8. REFERENCES


