Efficient SAT-based Bounded Model Checking for Software Verification

F. Ivančić[1], Z. Yang[1,2], M.K. Ganai[1], A. Gupta[1], and P. Ashar[1]

[1] NEC Laboratories America
4 Independence Way, Princeton, NJ 08540
[2] Western Michigan University, Dept. of Computer Science
Kalamazoo, MI 49008

Abstract. This paper discusses our methodology for formal analysis and automatic verification of software programs. It is currently applicable to a large subset of the `C` programming language that includes 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 modeling basic blocks. The program is then analyzed by a back-end SAT-based bounded model checker, where each unrolling is mapped to one step in a block-wise execution of the program.
The main contributions of this paper are as follows: 1) This paper is the first to use the block-based unrollings with SAT-based bounded model checking. This allows us to take advantage of SAT-based learning inherent to the best performing bounded model checkers. 2) We also present various heuristics used in the SAT-based bounded model checking customized for models automatically generated from software, allowing a more efficient analysis. 3) We have implemented our methodology into a prototype tool called F-Soft and applied it on various case studies. We present experimental results based on eight case studies including a `C`-based implementation of a network protocol, and compare the performance gains using the proposed heuristics.

1 Introduction

Although symbolic model checking algorithms using *binary decision diagrams* (BDDs) offer the potential of exhaustive coverage of large state-spaces, they often do not scale well in practice. An alternative is *bounded model checking* (BMC) [3] focusing on the search for counterexamples of bounded length only. The problem is translated to a Boolean formula, such that the formula is satisfiable iff there exists a counterexample of length k. In practice, k can be increased incrementally to find a shortest counterexample if it exists. However, additional reasoning is needed to ensure completeness of the verification when no counterexample exists [12, 16]. 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 [9, 13], SAT-based BMC can often handle much larger designs than BDDs.

Software modeling. In this paper, we describe our overall methodology for the analysis of software. Our prototype tool called F-Soft is developed for the `C`

programming language. 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 code. We perform this analysis via a translation to a circuit representation by considering the control and data flow of the program. The salient feature of our approach for software verification is the central role played by a basic block. From the modeling of software, the abstraction of the source code, to the verification of the generated software model using an unrolling based on blocks – the block modeling approach is integral to the proposed method. For each 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 (pc) consisting of $\lceil \log N \rceil$ bits is introduced to monitor progress in the state transition graph consisting of blocks.

The first key contribution of this paper is to use the pc variables to track progress of the allowed executions of the code during SAT-based BMC. This allows us to take advantage of SAT-based learning inherent to the best performing bounded model checkers. Effectively, an unrolling during BMC is understood to be one step in a block-wise execution of the program. In this context each atomic step of the BMC consists of a basic block rather than individual statements. This is an efficient way of performing BMC, since for each block there are 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, although each unrolling introduces all basic blocks into the satisfiability problem, many blocks in the new unrolling can be declared unreachable by merely considering the control flow of the software program. This intuition can be used to prune the search space considerably.

Comparison with CBMC. The most closely related work to ours is the CBMC tool [5], which also translates a `C` program into a Boolean representation to be analyzed by a back-end SAT-based BMC. However, there are many differences in our approaches. One major difference is that we generate a Boolean model of the software that can be analyzed by both SAT-based BMC and unbounded model checking using BDDs. Another major difference is the block-based approach, rather than a statement-based approach. Additionally, CBMC requires a full inlining and unwinding of the source code through a user-supplied constant. This allows the SAT-solver to consider only loop-free paths which is often fast for small pieces of code with no or few loops. However, this method of inlining and unwinding loops is clearly not scalable to larger pieces of code or to code with reactive behavior. Our translation method does not require multiple inlining or unwinding of loops, and is therefore more scalable than CBMC as shown in our experiments using F-Soft and CBMC presented in Section 5.

We also differentiate our approach through the use of pre-processing analyses such as *program slicing* and *range analysis*. A slice of a program with respect to a set of program elements is a projection of the program that includes only program elements that might affect the property [4]. Furthermore, we use range analysis techniques [15] to limit the number of bits needed to represent various

variables compared to using a full bit encoding as implemented in CBMC. We also do not need to rewrite the source code to single assignment form thus keeping only one copy of each variable. We also propose new heuristics to exploit many common features of Boolean models generated by our software modeling approach. We discuss three main heuristics which prune the SAT search space considerably, and the variations we have used for combining them. Finally, in contrast to CBMC, our tool also allows partial abstraction of the software using counterexample-guided predicate abstraction based on symbolic techniques similar to work described in [6]. However, we limit the discussion in this paper to our software modeling framework and our improved SAT-solver decision heuristics.

Verification problem. We define a *state* of a program to consist of a location $l \in L$ describing the current basic block, i.e. the pc variables, and a type-consistent evaluation of data variables where out-of-scope variables at location l are assigned an undefined value. We consider initial states of the program to be a single initial location, where each variable can take a random value that is type-consistent with its specification. The set of global variables includes a (bounded) fixed-length static variable modeling a function call stack, thus allowing modeling of *bounded recursion*. We currently focus on checking reachability in programs. For this, we define a set of locations Bad $\subseteq L$ to be unsafe, and our model checking analysis tries to prove or disprove whether these basic blocks can be reached. We have implemented our methodology in a prototype framework called F-Soft. At the back-end, it uses a Boolean analysis system called DiVer [10], which includes various SAT-based and BDD-based methods for performing both bounded and unbounded verification including BMC-based proof techniques for providing correctness proofs [16].

Overview. In Section 2 we discuss our block-based software modeling approach. Section 3 discusses our analysis using SAT-based BMC and proposes customized decision heuristics that improve the efficiency of the analysis. Section 4 introduces one of the analyzed network protocol case studies, while Section 5 presents experimental results for eight case studies using F-Soft in detail. Section 6 concludes this paper with some remarks and pointers to future work.

2 Basic Block Modeling and BMC

In this section, we describe our software modeling approach that is centered around basic blocks. 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 captures the control flow graph of the program. The data logic describes the assignments to variables. 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 (pc) variable is introduced to monitor progress in the control flow graph consisting of the basic blocks. The control logic defines the transition relation for the pc given the block transition graph and the conditions guarding the transitions between the blocks.

2.1 Control Logic Modeling

The key innovative contribution of this paper is to use the program counter to track progress of allowed executions of the code during BMC. The model checking analysis is performed by understanding an unrolling to be one step in a block-wise execution of the program, where each atomic step consists of a basic block rather than an individual statement. Similar approaches have been explored in a non-BMC setting to software model checking (such as with tools like VeriSoft [8], Java PathFinder [11, 18] and Bogor [14]), which have also extended it to deal with concurrency by using partial-order reduction. However, by incorporating this idea into a SAT-based BMC we are able to take advantage of recent progress in this research area, while also improving its efficiency for software verification by customizing the back-end SAT-solver.

We use certain code pre-processing steps to simplify the source code. For example, code simplification removes nested or embedded function calls inside other function calls by adding temporary variables. During the computation of the control flow graph (CFG), an edge from the calling block to the first block of the called function is created. If the function call returns a value, we add a statement assigning the return expression to the assigned variable. For functions that are not called recursively, 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, which is used to save the current local state and determine which basic block to return to. With respect to the subset of `C` that we currently consider, we support primitive data types, pointers, static arrays and records. As to dynamically created data structures such as dynamic arrays and linked-lists, an upper bound on the length is required and should be provided by the user. Furthermore, all control flow logic constructs are completely supported.

An example for our control logic modeling is shown in Figure 1, where the right side shows the computed CFG for the simple `C` code on the left side. The example shows how the basic blocks are computed for various types. Each basic block is identified by a unique number shown inside the hexagon adjacent to the basic block. The source node of the CFG is basic block 0, while the sink node is the highlighted basic block 8. The example in Figure 1 pictorially shows how non-recursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`.

2.2 Data Logic Modeling

Once the basic blocks of a `C` program are determined, we create the data logic for the assignments. We first simplify the assignments in each basic block, and

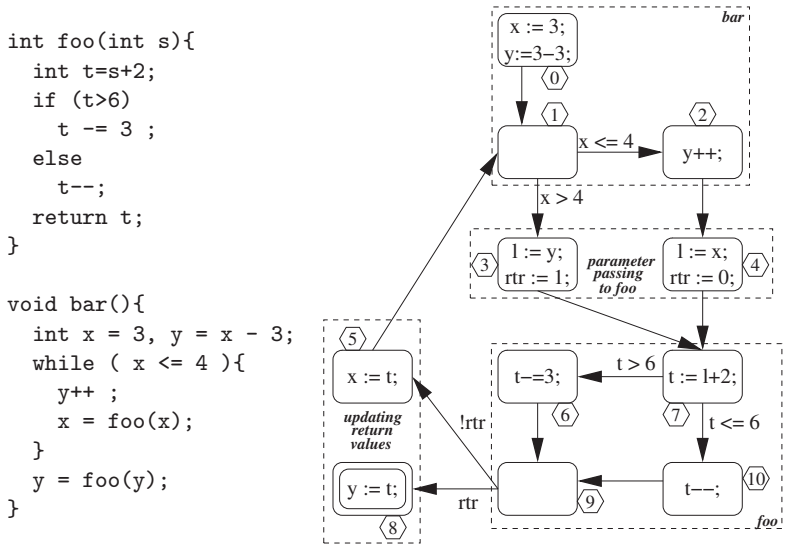

Fig. 1. Computing the control flow graph

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

Additional variables are introduced when pointer variables are declared since we are computing a Boolean representation. The declaration `int **p` creates three variables v_p, v'_p, v''_p, where v_p stands for `p`, v'_p for `*p`, and v''_p for `**p`. Similarly, `int *a, *b` creates four variables v_a, v'_a, v_b, v'_b, while a dereference in the `C` code, such as `&a`, also leads to additional variables – in this case the variable $'v_a$. Additional assignments have to be inferred due to aliasing and newly introduced variables. First, an assignment `p=&a` becomes $v_p = 'v_a$. Since `p=&a` implies `*p=a` and `**p=*a`, two new assignments $v'_p = v_a$ and $v''_p = v'_a$ are also inferred. An assignment `a=&x` gives rise not only to the assignment $v'_a = v_x$, but also to conditional assignments due to aliasing. Since `p` may equal `&a`, it is possible that `*p` and `**p` are assigned new values when `a` is assigned. This results in the conditional assignments `*p=(p==&a)?&x:*p` and `**p=(p==&a)?x:**p`. Some of the conditions in the conditional assignments can be removed based on previous assignments in the same basic block. In order to convert the sequential

assignments to parallel assignments, we also remove all possible read-after-write hazards through substitution. In addition, some assignments are redundant when considering a basic block as one atomic step. In particular, the assignments at later steps may overwrite previous assignments.

Next we consider how to convert the parallel assignments into a Boolean representation. All the variables after simplification and the range analysis procedure have finite domains. Assume we need t bits to represent a variable $\mathtt{var}$ with $\mathtt{var}_j (1 \leq j \leq t)$ being the current state bits and $\mathtt{var}'_j (1 \leq j \leq t)$ being the next state bits. Let $\mathtt{var}$ be assigned in blocks $\{b_1, b_2, \cdots, b_k\}$ and not assigned in the remaining blocks $\{b_{k+1}, \cdots, b_N\}$. The logic assigned to $\mathtt{var}_j$ is V_{ji} at block $b_i (1 \leq i \leq k)$. Also, let I_i be the index of the block b_i. The next-state data logic for $\mathtt{var}_j$ then is

$$\mathtt{var}'_j = (\bigvee_{i=1}^{k} (c_1 c_2 \cdots c_n = I_i) \wedge (V_{ji})) \vee (\bigvee_{i=k+1}^{N} (c_1 c_2 \cdots c_n = I_i) \wedge (\mathtt{var}_j)).$$

3 Model Analysis and Customized Decision Heuristics

The Boolean models generated by our software modeling contain many common features that are based on the particular translation presented here. Although we model the program counter to track progress in the control flow graph, there is additional information in the original software that could improve the efficiency of the SAT-based BMC. Although each unrolling introduces the whole code into the satisfiability problem, many blocks in the new unrolling can be declared unreachable by merely considering the control flow of the software program. In the following we propose various heuristics that improve the performance of the SAT-based BMC for the Boolean models generated from software. In section 5, we show these heuristics to be successful in our experiments.

3.1 Scoring of PC Variables

We adjust the decision heuristics used by the SAT-solver in 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 heuristic that increases the likelihood that the SAT-solver makes decisions first on variables that correspond to the control flow rather than the data flow, takes advantage of the fact that each new unrolling does not allow the whole code to be reached. We implement this heuristic by increasing the score for the bits of the pc variables, which in turn makes the back-end SAT-solver choose these variables as decision variables first. In addition to scoring pc variables higher than other variables in the system, we also control how to vary the scoring of variables over various time-frames.

3.2 One-Hot Encoding of Basic Blocks using Selection Bits

Another heuristic that we implemented is a one-hot encoding of the pc variables which allows the SAT-solver to make decisions on the full pc instead of the individual pc bits. In addition to the binary encoded pc variable already present in the circuit, we add a new selection bit for each basic block. The selection bit is set iff the basic block is active, i.e. when a certain combination of pc variable bits is valid. This provides a mechanism for word-level decisions since a certain basic block selection bit automatically invalidates all other selection bits through the pc variables. By increasing the score of the selection bits in comparison to other variables in the system, we are able to influence the SAT-solver to make quick decisions on the location first. An obvious disadvantage to this heuristic is that we need to add one selection bit to the Boolean model per basic block in the software code. However, this disadvantage is mitigated by the performance improvements as described in Section 5.

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

Let b_i represent the selection bit for basic block i and let $c_i, 0 \leq i \leq 3$ represent the four needed program counter variable bits for the code shown in Figure 1. We add constraints for each basic block, asserting the equality of the block selection variable with the binary-encoded pc label. For example, $b_0 = \bar{c_3}\bar{c_2}\bar{c_1}\bar{c_0}$ and $b_6 = \bar{c_3}c_2c_1\bar{c_0}$. Now, a decision by the SAT-solver to assign b_0 to 1 corresponds to a word-level decision on the program counter, which immediately results in implying $b_i = 0, i > 0$, including $b_6 = 0$ in our example. Furthermore, similar to the previously described technique of scoring variables in later time-frames higher than in prior time-frames, we also use this strategy for scoring of the selection bits.

3.3 Explicit Modeling of Incoming CFG Transitions

We can also aid the analysis of the back-end SAT-solver by constraining its search space to eliminate impossible predecessor-successor basic block combinations in the CFG. These constraints capture additional high-level information, which helps to prune the search space of the SAT-solver. At each depth, the choice of the SAT-solver to consider a particular basic block enables a limited number of possible predecessor blocks and eliminates immediately all other basic blocks from consideration. By increasing the likelihood that the SAT-solver decides first on the pc, we take advantage of the fact that each new unrolling does not allow the whole code to be reachable at each depth. We prefer to add these constraints based on incoming transitions into a basic block since the Boolean model of the software already encodes the outgoing transitions in terms of the pc variables.

This helps the SAT-solver since it provides bi-directional information about the transitions in the CFG.

For example, let c'_i, $0 \leq i \leq 3$, denote the next state pc variable bits in figure 1. The following constraint is added for basic block 7: $\bar{c'_3}c'_2c'_1c'_0 \rightarrow (\bar{c_3}\bar{c_2}c_1c_0 \vee \bar{c_3}c_2\bar{c_1}\bar{c_0})$, to reflect the constraint that if the current block is 7, then the predecessor block is either 3 or 4. Similarly, for basic block 3, we add the following constraint: $\bar{c'_3}\bar{c'_2}c'_1c'_0 \rightarrow (\bar{c_3}\bar{c_2}\bar{c_1}c_0 \wedge g)$, where g stands for the representation of `x>4`. We can also add such constraints in terms of the one-hot encoded selection bits of the basic blocks if both these heuristics are being utilized. The constraint for basic block 7 then simply becomes $b'_7 \rightarrow (b_3 \vee b_4)$.

We also allow the user to customize this heuristic by choosing a subset of basic blocks for which to apply this heuristic. This customization allows the user to fine-tune the amount of additional constraints generated, since too many additional constraints may burden the SAT-solver rather than improve efficiency. In addition, another optimization, that we have not yet implemented, is to limit the additional constraints to basic blocks at depths that are actually reachable at that depth. This optimization, similar to the optimization described in the one-hot encoding heuristic, requires an additional BFS of reachable basic blocks.

4 Network Case Study: Point-to-Point Protocol

The verification of various network protocols has been a popular application of model checkers [7]. Most of these approaches have analyzed models of network protocols with respect to some temporal properties such as absence of deadlock where the model is based on a textbook description or an RFC (Request for Comments) document. In addition to this work, there have recently been attempts to verify an implementation of a protocol with respect to its standard defined in an RFC [2]. In this section, we describe the Point-to-Point protocol, which is one of the case studies that we have analyzed using F-Soft.

The Point-to-Point Protocol (PPP) provides a standard method for transporting multi-protocol datagrams over point-to-point links. PPP contains three main components, namely a method for encapsulating multi-protocol datagrams, a Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection, and a family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols. Similar to [2], we focus on checking the implementation of the option negotiation automaton of the LCP part of PPP for link establishment.

RFC 1661 [17] 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 1 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.

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

Table 1. A part of the PPP specification

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

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

For the purpose of this paper, we edit the public implementation to be limited to LCP. We want to verify that the public implementation adheres to the specification as it is given in RFC 1661. The current prototype implementation of our verification engine does not allow verification of multi-threaded code. We implemented the following scheme to analyze the implementation of PPP which is illustrated in Figure 2: To check the public implementation we need a second peer to engage it. We developed an environment peer based on the RFC in C, that also initiates requests non-deterministically. Since our goal is to analyze whether the public implementation adheres fully to the RFC specification, we also implemented a specification model of the RFC in C.

The environment and the public implementation are engaged in communication, while the specification "listens in" on messages received by the public implementation. Although the specification tries to respond to these messages, its outgoing link is "grounded" and does not affect the environment peer. Our

analysis thus translates to the question whether the state of the public implementation always matches the state of the RFC-based specification.

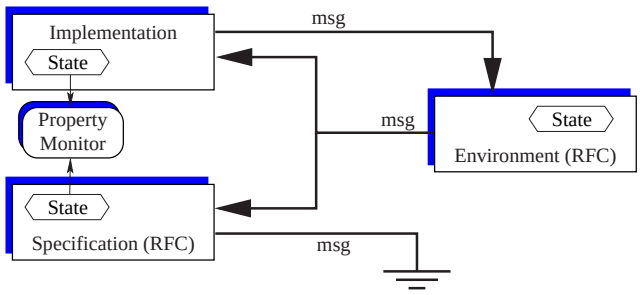

Fig. 2. PPP Verification Setup

In [2], the C-program as described here, has been manually translated to the input language of the model checker MOCHA[1]. 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 [2] 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. We present experimental results for this case study in more detail in the following section.

5 Experimental Results

We have performed various case studies comparing the various customized decision heuristics using F-SOFT. In the following we describe the results of eight case studies that we have recently completed. In all experiments we assume a 4-hour time limit for the analysis. For the first five case studies named W1–W5 the analysis finds that the stated property is incorrect and a *witness* trace exists, while for the later three case studies named P1–P3 the analysis finds a *proof* for the property. We summarize the experimental results in Table 2. W3 is the aforementioned PPP network protocol case study described in Section 4, while W1 and W2 are related pieces of software of similar complexity. The designs W4 and W5 are based on the same source code but consider different properties. The code for W4 and W5 is relatively small and is purely straight-line code with no loops. The provable case studies P1 and P2 are based on the same software describing a simple reactive and interleaved mutual exclusion example where we use different proof strategies to derive the correctness proof. P3 is an unrelated simple code fragment based on interface specification checking.

Results for customized heuristics used in BMC. For each case study, Table 2 includes the number of latches in the circuit representation in the second

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

Table 2. Experimental Results

column. We include two numbers, where the first number corresponds to the number of latches when the one-hot encoding heuristic is not used, while the second number represents the number of latches with one-hot encoding. Please note that we use the aforementioned range analysis techniques [15] to limit the number of bits per variable. We also compute the cone of influence of the circuit model with respect to the property which further reduces the number of latches to be considered. The next column contains the number of gates considered for these case studies, where the range depends on the subset of heuristics used for the various verification runs. The lower bound corresponds to the case that the standard DiVer is used, while the upper bound represents the number of gates considered when all heuristics are used. The following column labeled **in** denotes the number of primary inputs used for the analysis of the respective property.

The following columns represent the experimental results with respect to CPU time measured in seconds for our BMC analyses using the various combinations of heuristics. The column labeled **std** represents the case when we use standard DiVer without any of the customized heuristics described here. It can be seen that the standard DiVer implementation is not able to complete the analysis for four of the eight designs due to a time-out, denoted TO, where the 4-hour time-limit expires. The following columns give the experimental results for various combination of heuristics used for the analysis. We abbreviate the heuristics *scoring of pc variables* by **sc**, *one-hot encoding* by **1h** and *explicit modeling of CFG transitions* by **tr**. For example, the column **1h&tr** corresponds to the combination of one-hot encoding and explicit modeling of CFG transitions, while **all** includes additionally the scoring of pc variables.

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

When considering the possible combinations of heuristics, it is noteworthy that for designs W1–W5 the advantage of using the one-hot encoding is supported by the addition of other heuristics. A similar statement can however not be made for the provable designs P1–P3 where the results are currently inconclusive with respect to the best combination of heuristics. For P1 we find that using scoring of pc variables with explicit modeling of CFG transitions improves the performance somewhat over the case that only the explicit modeling of transitions is used. In summary, we find that in most cases a combination of one-hot encoding of basic blocks with explicit modeling of incoming transitions is a good default choice when the BMC searches for both a proof and a witness. During standard witness searches, using one-hot encoding only seems effective.

Results using BDDs. In addition to the results presented in Table 2, we have also used our BDD-based model checker on the generated models. For designs W1, W2, W4 and W5 our BDD-based model checker was able to find a witness trace, albeit being slower than BMC and taking up to four times the amount of CPU time. In the case of W3, the BDDs were not able to complete the analysis in the given time-bound. For designs P1–P3, BDDs were able to complete the analysis in all cases rather quickly due to the small model sizes. For these examples, the BDD-based model checker beat any of the BMC-based analyses given in Table 2. However, our experience in hardware verification suggests that SAT-based proof techniques become increasingly important compared to BDDs when considering larger provable models. Our current results provide preliminary evidence for the effectiveness of our customized SAT heuristics. In the future we hope to try our technique on larger provable case studies.

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

Detailed results for PPP. Figures 3 and 4 show a more detailed comparison of the individual heuristics and their performance for design W3. Figure 3 shows the cumulative time in seconds taken for all depths up to a given number, while Figure 4 shows the respective time each verification step 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 a SAT-based BMC for the analysis, since the standard includes various peaks in the computation time, in particular for depths 20-35, but better performance afterwards. That indicates that the SAT-solver is able to learn important invariants of the design early on that enable a deeper analysis later.

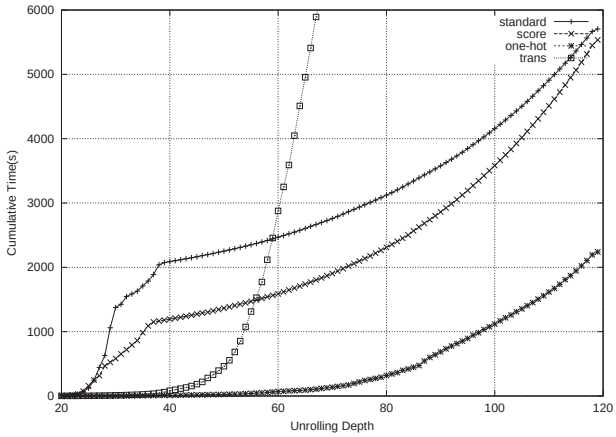

Fig. 3. Cumulative time comparison of BMC heuristics for design W3

The figures also include three more graphs each, describing the respective performance of BMC using the heuristics scoring of pc variables (*score*), one-hot encoding (*one-hot*) and encoding of CFG transitions (*trans*) individually. It is noteworthy that the inherent learning in the SAT-solver is preserved, which is visible by the various peaks in the respective graphs. Figure 3, in particular, shows the advantage of the one-hot encoding heuristic for the design W3 which consistently outperforms the other heuristics.

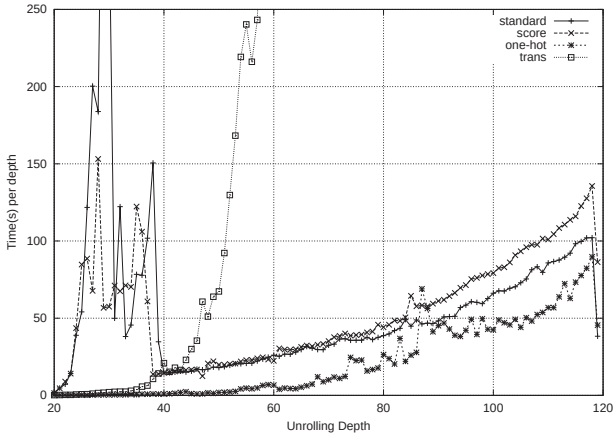

Fig. 4. Time per depth comparison of BMC heuristics for design W3

Experience with CBMC. We also used the CBMC tool to run the analysis for the same examples. Our experience with CBMC was somewhat mixed. With respect to applicability of the same source code, the needed transformations

were small. They involved changing the name of the function used to model nondeterminism, removal of certain header file information that were included during our code preprocessing and simplifications, which were not supported by CBMC, moving certain variable declarations outwards to avoid redeclaration during multiple unwindings, changing some goto statements with forward jumps, as well as explicitly changing all fall-through statements in switch statements to recopy the code as necessary since CBMC does not allow fall-through.

During our initial experiments we found that CBMC was able to analyze various small code fragments rather quickly, especially if the code did not include any loops. For example, for the small example of straight-line code used in designs W4 and W5, CBMC was faster than F-Soft. This is due to the fact that CBMC generates a single SAT formula for all execution paths which is feasible for such small examples, and can then be analyzed efficiently for small examples by the back-end SAT-solver. While our tool can perform such an analysis as well, we only report results for incremental BMC runs using F-Soft here. For the PPP example (W3) described in section 4 however, CBMC was not able to unwind the software even once, and instead ran out of memory. One unwinding would also not have been enough to produce the counterexample. For the example P1 and P2, we were able to unwind the small software piece enough times, however the zChaff [13] SAT-solver included in CBMC was not able to complete the analysis in the provided 4 hour time-limit given only one large SAT formula. For the P3 example we were not able to parse it successfully using CBMC since P3 is a case study written for Microsoft Windows.

6 Conclusions and Future Work

This paper describes the customized SAT-based heuristics we have used for efficient analysis of models generated by our software verification methodology. This methodology is founded upon block software modeling, BMC-unrolling and symbolic predicate abstraction. It is currently applicable for a large 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 translate a program into a Boolean representation to be analyzed by a back-end SAT-based BMC, by interpreting an unrolling during BMC as one step in a block-wise execution of the program. We have included new SAT-based heuristics in our verification engine DiVer, which are able to take advantage of the fact that we are considering a design automatically abstracted and generated from software. Additionally, we presented detailed experimental results using our customized heuristics for eight case studies.

There remain many research directions to follow in the future. We are currently implementing a counterexample-guided predicate abstraction method using symbolic techniques. In addition, we are studying the applicability of the tool for verification of concurrent `C` programs. For this scenario, we will use the feature that we are able to split basic blocks into smaller basic blocks to ensure proper concurrent interleaving semantics based on the statements and the scope

of variables in the considered blocks. Our initial results for various case studies are showing promising directions for future research.

References

1. R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Intern. Conf. on Software Engineering*, pages 835–836, 2001.
2. R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Computer Aided Verification*, 2001.
3. A. Biere, A. Cimatti, E.M. 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*, pages 317–320, 1999.
4. E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Programs slicing for VHDL. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):125–137, October 2002.
5. E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.
6. E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proc. of the Model Checking for Dependable Software-Intensive Systems Workshop, San-Francisco, USA*, 2003.
7. J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proc. of Intern. Conf. on Computer-Aided Verification*, LNCS 1102. Springer, 1996.
8. P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Computer-Aided Verification*, volume 1254 of *LNCS*. Springer, 1997.
9. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
10. A. Gupta, M.K. Ganai, C. Wang, Z. Yang, and P. Ashar. Abstraction and BDDs complement SAT-based BMC in DiVer. In *15th Intern. Conf. on Computer Aided Verification*, volume 2725 of *LNCS*, pages 206–209. Springer, 2003.
11. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), April 2000.
12. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *4th Intern. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 298–309. Springer, January 2003.
13. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
14. Robby, M.B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *9th European software engineering conference*, pages 267–276. ACM Press, 2003.
15. R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.
16. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 108–125. Springer, November 2000.
17. W. Simpson. PPP: The Point-to-Point Protocol. RFC 1661, IETF, June 1994.
18. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th IEEE Intern. Conf. on Automated Software Engineering*, 2000.