Efficient SAT-based Bounded Model Checking for Software Verification

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

[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.
In this paper we first briefly review our method that uses block-based unrollings with SAT-based bounded model checking for software verification as presented in [15]. This allows us to take advantage of SAT-based learning inherent to the best performing bounded model checkers. In contrast to [15], this paper then focusses on various heuristics used in the SAT-based bounded model checking customized for models automatically generated from software, allowing a more efficient analysis. We have implemented our methodology into a prototype tool called F-Soft and applied it to various case studies using the proposed heuristics. We present experimental results based on six case studies 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 [17, 23]. 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 [11, 19], SAT-based BMC can often handle much larger designs than BDDs.

Software modeling. In this paper, we first briefly review our overall methodology for the automatic analysis of software programs as discussed in [15]. Our prototype tool is developed for the `C` programming language. Similar to work described in [8], 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 representation by considering the control and data flow of the program. In contrast to [8], our procedure is based on translating blocks instead of individual statements as atomic components of programs. Furthermore, they focus on generating a monolithic SAT formula for checking equivalence of a `C`-program with a register-transfer level (RTL) hardware design, whereas we focus on generating SAT problems for BMC by iteratively unrolling a block-based execution 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 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 (pc) consisting of $\lceil \log N \rceil$ bits is introduced to monitor progress in the state transition graph consisting of basic blocks. As described in [15], we use the pc variables to track progress of the allowed executions of the source code during SAT-based BMC. Effectively, an unrolling during BMC is understood to be one step in a block-wise execution of the program.

This is an efficient way of performing BMC, since for each basic block there are only a limited number of possible successors. Given a single initial block label, there are thus only a limited number of possible next blocks reachable in new unrollings. Thus, 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. One key contribution of this paper is to use this intuition to prune the search space considerably by proposing heuristics that allow an efficient SAT-based BMC of the generated models. In contrast to [15], we further give detailed experimental results for six case studies using the proposed heuristics.

The Boolean models generated by this software modeling approach contain many common features that can be exploited by the back-end SAT-solver. In this paper, we propose three particular heuristics that are able to prune the search space of the SAT-solver considerably. The three heuristics are to increase the relevance of pc variables to the SAT-solver by higher scoring of pc variables, a one-hot encoding of basic blocks using one selection bit per basic block, and an explicit modeling of incoming transitions to basic blocks in the control flow graph (CFG). We present these heuristics in more detail in section 3.

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 in a single 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 [12], which includes various SAT-based and BDD-based methods for performing both bounded and unbounded verification.

Overview. In section 2 we discuss our block-based software modeling approach as it has been presented in more detail in [15]. Section 3 then discusses our model analysis using SAT-based BMC and proposes new customized decision heuristics that improve the efficiency of the analysis. Section 4 discusses briefly our prototype tool implementation F-SOFT, while section 5 presents experimental results for various 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 review our software modeling approach that is centered around basic blocks as it has been detailed in [15]. 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. 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 the pc variables given the block transition graph and the conditions guarding some of the transitions between the blocks. In the following we discuss our software modeling approach, considering first the control flow and then the data flow.

2.1 Control Logic Modeling

In [15], we described how 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 [10], Java PathFinder [13, 24] and Bogor [21]), 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.

This modeling and analysis approach is an efficient way of performing BMC, since for each basic block and thus also for each unrolling of a block there is only a limited number of possible successors. Given a single initial block label, there is thus only a limited number of possible next blocks reachable in new unrollings. 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.

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 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 determine which basic block to return to once the computation is completed. A guard is added on the returning transition which checks the information stored on the function call stack.

The right side of figure 1 shows the computed CFG using our approach 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 an unique number shown inside the hexagons adjacent to the basic blocks. 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 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`.

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 then create a Boolean representation. In order to obtain the logic for each variable assignment `var=expr`, we then build combinational circuits 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 the circuits for `expr1` and `expr2`. The result has the same bit-width as `vec1` and `vec2`, and each result bit is the output of an AND gate

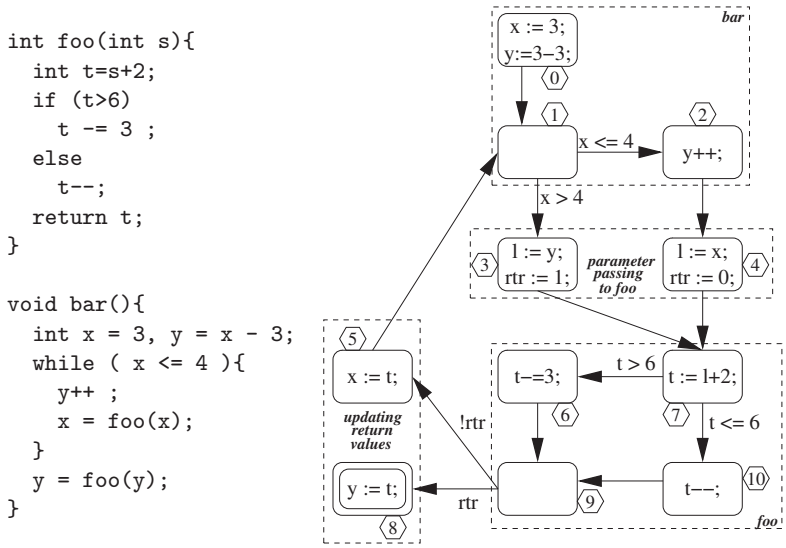

Fig. 1. Computing the control flow graph

with two inputs being the corresponding bits in `vec1` and `vec2`. To handle an expression of type `expr1+expr2`, we create a bull adder for each bit. For the case of an relational expression the result has only one bit. In the following we briefly review the main differences in our methodology from previous approaches such as [8]. The differences are mainly centered around assignment simplifications due to basic block modeling that allow the sequential assignments in a basic block to be treated as parallel after the code simplification.

Additional variables are introduced when pointer variables are declared as 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 assignment $v'_p = v_a$ and $v''_p = v'_a$ are 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 be equal to `&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 se-

quential assignments to parallel assignments, we also remove all possible read-after-write hazards. By doing so, we replace the read variables on the right hand side with the assigned value if the same variables were assigned previously in the same basic block. 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 the software modeling contain many common features that are based on the particular translation as proposed in [15] and reviewed 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. This modeling and analysis approach is an efficient way of performing BMC, 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 time-frames. 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 Increased Relevance of PC Variables

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 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 based on a static analysis of the control flow graph. 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. We also use the terminology *scoring of pc variables* to refer to this heuristic. This heuristic forces the BMC to focus first on a static reachability computation of the CFG and is thus able to eliminate many traces quickly.

In addition to scoring pc variables higher than other variables in the system, we are also able to control how to vary the scoring of variables over various time-frames. For example, it turns out that increasing the relevance of pc variables more in later time-frames than in previous ones takes advantage of the SAT-solver decisions inherent in our DiVer [12] tool.

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. 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 basic block selection bits through the pc variables. By increasing the score to set a basic block selection bit compared 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 include one selection bit to the Boolean model per basic block in the software code.

In addition, we are able to increase the score intelligently for these basic block selection bits by considering the pre-computed static reachability information from the CFG. We only need to increase the score for 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 on the CFG.

Each basic block in figure 1 is given an unique number which will be used to describe the one-hot encoding. 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. We add constraints for each basic block, asserting the equality of the block selection variable with the binary-encoded program counter label. For example, $b_0 = \bar{c_3}\bar{c_2}\bar{c_1}\bar{c_0}$ and $b_6 = \bar{c_3}c_2c_1\bar{c_0}$. Now, a decision by the SAT-solver to assign b_0 to 1 corresponds to a word-level decision on the program counter, which immediately results in implying $b_i = 0, i > 0$, including $b_6 = 0$ in our example.

Furthermore, similar to the previously described advantage to score variables in later time-frames higher than in prior time-frames, we are also able to use the same strategy for the scoring of the selection bits described in this heuristics.

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.

For example, assuming that c'_i with $0 \leq i \leq 3$ denote the next state pc variable bits, the following constraint is added for basic block 7 of figure 1: $\bar{c'_3}c'_2c'_1c'_0 \rightarrow (\bar{c_3}\bar{c_2}c_1c_0 \vee \bar{c_3}c_2\bar{c_1}\bar{c_0})$. 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 $b'_7 \rightarrow (b_3 \vee b_4)$. With the addition of this constraint, if the SAT-solver assigns b'_7 to 1, i.e. the current block is 7, then the predecessor block is required to be one of the blocks 3 or 4.

We 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. We currently allow the user to specify one of the following three choices if this heuristic is applied:

- Include additional constraints for all basic blocks in the CFG.
- Include additional constraints only for those basic blocks that have multiple incoming transitions. This option requires the SAT-solver to pick one incoming edge when there are several choices. This allows the SAT-solver to quickly focus on a single feasible path at a time.
- Include additional constraints only for those basic blocks that have a single incoming transition. This option helps the SAT-solver to quickly propagate a path backwards when there are no multiple incoming edges. This process alleviates unnecessary decisions on pc variables immediately in such basic blocks.

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 The F-Soft Tool

We have implemented our methodology in a prototype framework called F-Soft. The input to F-Soft is a set of files corresponding to a software module represented as a `C` program. Furthermore, the input also consist of the property

to be checked which is specified by a property monitor. For now, we assume that the property is specified by referring to a labeled basic block in the source code. We also differentiate our approach from other BMC-based approaches through the use of pre-processing analyses such as *program slicing* [6, 9, 16] and *range analysis* [4, 5, 20, 22], which are used to limit the number of bits needed to represent various statements in the code. This is in contrast to [8], where for each statement (at least) 32 bits are used. 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 using the pc variables. Additionally, we include a counterexample-guided predicate abstraction method based on symbolic techniques in our tool.

Abstraction is probably the most important technique for reducing the state explosion problem [7]. In contrast to initial work on predicate abstraction (such as [1, 14]), 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 in our F-Soft tool. A similar approach has recently been advocated and shown advantageous over the use of theorem provers [18]. As discussed in [15], we allow a *partial abstraction* of the software using predicates thus yielding a model that contains both abstracted as well as concrete data variables. This allows a seamless tradeoff between accuracy and ease of analysis, potentially enabling accurate analysis with a fewer number of abstraction-refinement iterations. Additionally, we proposed various optimizations for the computation of the abstract transition relation in [15].

Our Boolean verification framework called DiVer [12] uses various SAT-based and BDD-based methods for performing both bounded and unbounded verification. As we have discussed in detail in the previous section, 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. 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 [2]) or a SAT-based approach (such as [8]).

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. However, if DiVer finds a counterexample in the abstracted Boolean model, and the analysis of this counterexample finds that the path is actually feasible in the original source code, the tool has discovered a concrete counterexample in the original system. We are currently implementing a counterexample-guided refinement method based on symbolic techniques.

5 Experimental Results for Customized Heuristics

We have performed various case studies comparing the various customized decision heuristics using the F-Soft tool. In the following we describe the results of six case studies that we have recently completed. In all experiments we assume a

4-hour time limit for the analysis. For the first three case studies named W1–W3 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 that the property is correct. We summarize the experimental results in table 1. In this paper we only consider the effect of the here proposed heuristics and do not use our implemented predicate abstraction method.

	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
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 1. Experimental Results

For each case study, table 1 includes the number of latches in the cone of influence of the property to be verified in the second column. We include two numbers, where the first number corresponds to the amount of latches when the one-hot encoding heuristic is not used, while the second number represents the amount of latches in the cone of influence with the one-hot encoding of basic blocks. 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 SAT-solver is used, while the upper bound represents the amount 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 used. 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 the two case studies W1 and W2 due to a time-out represented by 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 shows that using just the one-hot encoding heuristics proves very efficient for the analysis of the designs W1–W3 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–W3 when used individually.

When considering the possible combinations of heuristics, it is noteworthy that for the designs W1–W3 the advantage of using the one-hot encoding is supported by the addition of the other heuristics. A similar statement can however not be made for the results for designs P1–P3 which 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. Interestingly, we note that in most cases, using the explicit modeling of transitions only for basic blocks with multiple incoming edges beats the other options. However, this result is, as of now, still inconclusive and will need further experimental evidence in the future. In summary, however, 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 of heuristics to be used.

In addition to the results presented in table 1, we have also used our BDD-based model checker on the generated models. For the designs W1 and W2 we have found that our BDD-based model checker was able to find a witness trace, albeit taking up to four times the amount of CPU time. In the case of W3, our BDD-based model checker was actually not even able to complete the analysis in the given time-bound. For the case studies P1–P3, our BDD-based model checker was able to complete the analysis in all cases rather quickly given the small model sizes. In all examples, P1–P3, the BDD-based model checker beat any of the BMC-based analyses given in table 1. In the future we want to see how the heuristics and the BDD-based model checker perform for larger provable case studies.

Figures 2 and 3 show a more detailed picture for the comparison of the individual heuristics and their performance for design W3. Figure 2 shows the cumulative time in seconds taken for all depths up to that point, while figure 3 shows the respective time each verification step for a particular depth given the same heuristics. The graphs labeled *standard* represent the standard decision heuristics implemented in the DiVer tool. The graph clearly shows the advantage of using a SAT-based BMC for the analysis, since the standard heuristics 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.

The figures also includes three more graphs each, describing the respective performance of the 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 2, in particular,

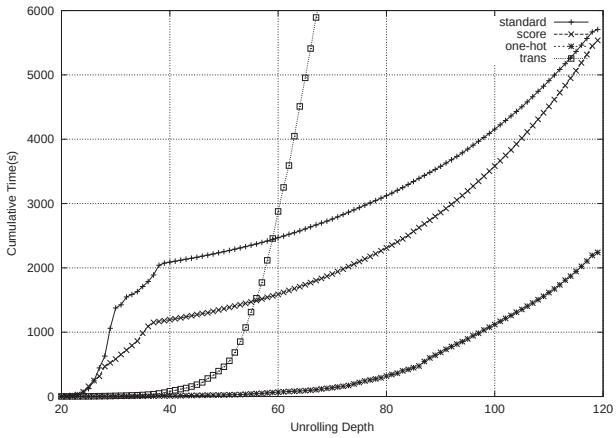

Fig. 2. Cumulative time comparison of BMC heuristics for design W3

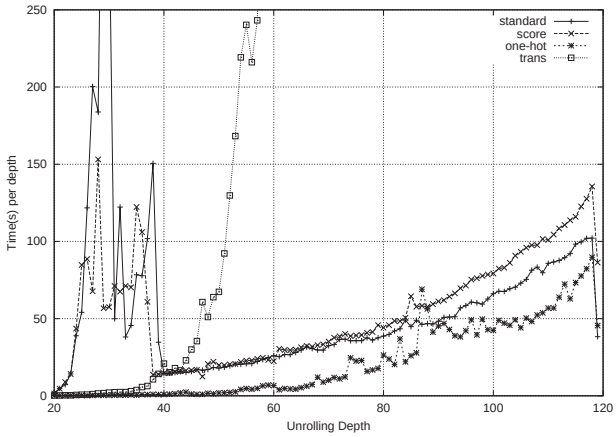

Fig. 3. Time per depth comparison of BMC heuristics for design W3

shows the advantage of the one-hot encoding heuristic for the design W3 which consistently outperforms all other heuristics.

Figures 4 and 5 compares the various choices for the explicit modeling of CFG transitions as described earlier for design P2. In this particular design, using this heuristic only for basic blocks that contain multiple incoming edges outperforms all other choices. The corresponding graph in the figures is labeled *multiple*. Using the heuristics for those basic blocks that have only one incoming edge (*single*), performs worst in this design. However, it should be noted that our conclusions about the subset of basic blocks to be used for this heuristics is not complete yet, and we are planning to study the performance impact on more case studies. Again, it can be seen that the inherent learning in the SAT-solver is preserved which justifies our approach of SAT-based BMC for models generated and abstracted from software.

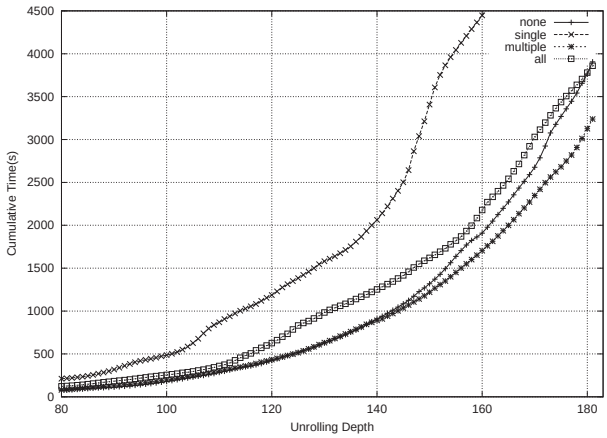

Fig. 4. Cumulative time comparison for incoming transitions heuristic using design P2

6 Conclusions and Future Work

This paper introduces our software verification methodology founded upon basic 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 the BMC as one step in a block-wise execution of the program.

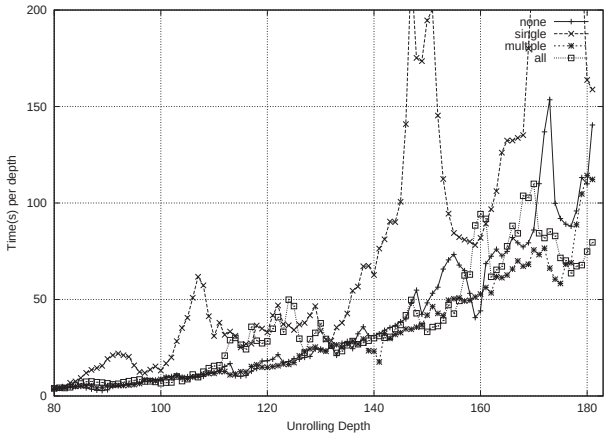

Fig. 5. Time per depth comparison for incoming transitions heuristic using design P2

We are able to adjust and include new decision 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. We have proposed customized SAT-based BMC heuristics for models generated from software that allow an efficient analysis for these models. Additionally, we presented initial experimental results using our customized heuristics for six case studies.

There remain many research directions to follow in the future. We are currently implementing a counterexample-guided predicate abstraction method based on symbolic techniques. In addition, we are studying the applicability of the tool for verification of concurrent `C` programs. However, initial results for various case studies are showing promising directions for future research.

References

1. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
2. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.
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. R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.
5. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Trans. Program. Lang. Syst.*, 21(4):747–789, 1999.

6. 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.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 2000.
8. 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. To appear.
9. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448. 2000.
10. P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the Ninth International Conference on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*. Springer, 1997.
11. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
12. 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.
13. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), April 2000.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
15. F. Ivančić, M. Ganai, A. Gupta, P. Ashar, and Z. Yang. Modeling, abstraction and customized heuristics for SAT-based bounded model checking of software. 2004. (submitted).
16. J. Krinke. Context-sensitive slicing of concurrent programs. In *9th European software engineering conference*, pages 178–187. ACM Press, 2003.
17. 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.
18. S.K. Lahiri, R.E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification*, volume 2725 of *LNCS*. Springer, 2003.
19. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
20. J.R.C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995.
21. 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.
22. 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.
23. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, November 2000.
24. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, 2000.