SMT-Based Bounded Model Checking for Embedded ANSI-C Software

Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva

Abstract—Propositional bounded model checking has been applied successfully to verify embedded software but remains limited by increasing propositional formula sizes and the loss of high-level information during the translation preventing potential optimizations to reduce the state space to be explored. These limitations can be overcome by encoding high-level information in theories richer than propositional logic and using SMT solvers for the generated verification conditions. Here, we propose the application of different background theories and SMT solvers to the verification of embedded software written in ANSI-C in order to improve scalability and precision in a completely automatic way. We have modified and extended the encodings from previous SMT-based bounded model checkers to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers. We have integrated the CVC3, Boolector, and Z3 solvers with the CBMC front-end and evaluated them using both standard software model checking benchmarks and typical embedded software applications from telecommunications, control systems, and medical devices. The experiments show that our ESBMC model checker can analyze larger problems than existing tools and substantially reduce the verification time.

Index Terms—Software engineering, formal methods, verification, model checking.

1 Introduction

Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) has been introduced as a complementary technique to Binary Decision Diagrams (BDDs) for alleviating the state explosion problem [1]. The basic idea of BMC is to check the negation of a given property at a given depth: given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a verification condition (VC) ψ such that ψ is satisfiable if and only if ϕ has a counterexample of depth k or less. Standard SAT checkers can be used to check whether ψ is satisfiable. In BMC of software, the bound k limits the number of loop iterations and recursive calls in the program.

In order to cope with increasing software complexity, SMT (Satisfiability Modulo Theories) solvers can be used as back-ends for solving the generated VCs [2], [3], [4], [6]. Here, predicates from various decidable theories are not encoded using propositional variables as in SAT, but remain in the problem formulation. These theories are handled by dedicated decision procedures. Thus, in SMT-based BMC, ψ is a quantifier-free formula in a decidable subset of first-order logic which is then checked for satisfiability by an SMT solver.

- *L. Cordeiro is with the Electronic and Information Research Center, Federal University of Amazonas, Brazil. E-mail: lucascordeiro@ufam.edu.br*
- *B. Fischer is with the School of Electronics and Computer Science, University of Southampton, United Kingdom, SO17 1BJ. E-mail: b.fischer@ecs.soton.ac.uk*
- *J. Marques-Silva is with the Dept. of Computer Science and Informatics, University College Dublin, Ireland, and IST/INESC-ID, Lisbon, Portugal. E-mail: jpms@ucd.ie*

In order to reason about embedded software accurately, an SMT-based BMC must consider a number of issues that are not easily mapped into the theories supported by SMT solvers. In previous work on SMT-based BMC for software [2], [3], [4] only the theories of uninterpreted functions, arrays and linear arithmetic were considered, but no encoding was provided for ANSI-C [5] constructs such as bit-level operations, fixed-point arithmetic, pointers (i.e., pointer arithmetic and comparisons) and unions. This limits its usefulness for analyzing and verifying embedded software written in ANSI-C. In addition, the SMT-based BMC approaches proposed by Armando et al. [2], [3] and by Kroening [6] do not support the checking of arithmetic overflow and do not make use of high-level information to simplify the unrolled formula. We address these limitations by exploiting the different background theories of SMT solvers to build an SMT-based BMC tool that precisely translates program expressions into quantifier-free formulae and applies a set of optimization techniques to prevent overburdening the solver. This way we achieve significant performance improvements over SAT-based BMC and the previous work on SMT-based BMC [2], [3], [4], [6].

Our work makes two major contributions. First, we describe the details of an accurate translation from single-threaded ANSI-C programs into quantifier-free formulae using the logics QF_AUFBV and QF_AUFLIRA from the SMT-LIB [9]. Second, we demonstrate that our encoding and optimizations improve the performance of software model checking for a wide range of software systems, with a particular emphasis on embedded software. Additionally, we show that our encoding allows us to reason about arithmetic overflow and to verify programs

that make use of bit-level, pointers, unions and fixed-point arithmetic. We also use three different SMT solvers (Boolector [17], CVC3 [16], and Z3 [18]) in order to check the effectiveness of our encoding techniques. We considered these solvers because they were the most efficient ones for the categories of QF_AUFBV and QF_AUFLIRA in the last SMT competitions [10]. To the best of our knowledge, this is the first work that reasons accurately about ANSI-C constructs commonly found in embedded software and extensively applies SMT solvers to check the VCs emerging from the BMC of industrial embedded software applications. We implemented our ideas in the ESBMC[1] (Efficient SMT-Based Bounded Model Checker) tool that builds on the front-end of the C Bounded Model Checker (CBMC) [11], [30]. ESBMC supports different theories and SMT solvers in order to exploit high-level information to simplify and to reduce the formula size. Experimental results show that our approach scales significantly better than both the SAT-based and SMT-based CBMC model checker [11], [30], [6] and SMT-CBMC [3], a bounded model checker for C programs that is based on the SMT solvers CVC3 and Yices.

This paper extends our previous work [7], [8]. The version of ESBMC described and evaluated here has been optimized and extended. It now includes checks for memory leaks and a generic SMT-LIB backend, in addition to the native backends for Boolector, CVC3, and Z3. We have also significantly expanded the experimental basis and evaluate ESBMC with (resp. compare it against) the most recent stable versions of the SMT solvers and BMC tools. The remainder of the paper is organized as follows. We first give a brief introduction to the CBMC model checker and describe the background theories of the SMT solvers that we will refer throughout the paper. In Section 3 we present an accurate translation from ANSI-C programs into quantifier-free formulae using the SMT-LIB logics and explain our approach to exploit the different background theories and solvers. In Section 4 we present the results of our experiments using several software model checking benchmarks and embedded systems applications. In Section 5 we discuss the related work and we conclude and describe future work in Section 6.

2 Background

ESBMC builds on the front-end of CBMC to generate the VCs for a given ANSI-C program. However, instead of passing the VCs to a propositional SAT solver, ESBMC converts them using different background theories and passes them to an SMT solver. In this section, we describe the main features of CBMC that we use, and present the background theories used.

2.1 C Bounded Model Checker

CBMC implements BMC for ANSI-C/C++ programs using SAT solvers [11]. It can process C/C++ code using the goto-cc tool [12], which compiles the C/C++ code into equivalent GOTO-programs (i.e., control-flow graphs) using a gcc-compliant style. The GOTO-programs can then be processed by the symbolic execution engine. Alternatively, CBMC uses its own, internal parser based on Flex/Bison, to process the C/C++ files and to build an abstract syntax tree (AST). The typechecker of CBMC's front-end annotates this AST with types and generates a symbol table. CBMC's IRep class then converts the annotated AST into an internal, language-independent format used by the remaining phase of the front-end.

CBMC uses two recursive functions that compute the *constraints* (i.e., assumptions and variable assignments) and *properties* (i.e., safety conditions and user-defined assertions). In addition, CBMC automatically generates safety conditions that check for arithmetic overflow and underflow, array bounds violations, and NULL-pointer dereferences, in the spirit of Site's clean termination [13]. Both functions accumulate the control flow predicates to each program point and use these predicates to guard both the constraints and the properties, so that they properly reflect the program's semantics. CBMC's VC generator (VCG) then derives the VCs from these.

Although CBMC implements several state-of-the-art techniques for propositional BMC, it still has the following well-known limitations [3], [4]: *(i)* large data-paths involving complex expressions lead to large propositional formulae due to the number of variables and the width of data types, *(ii)* the loss of high-level information during the translation prevents potential optimizations to prune the state space to be explored, and *(iii)* the size of the encoding increases with the size of the arrays used in the program.

2.2 Satisfiability Modulo Theories

SMT decides the satisfiability of first-order formulae using a combination of different background theories and thus generalizes propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories. Given a theory $\mathcal{T}$ and a quantifier-free formula ψ, we say that ψ is $\mathcal{T}$-satisfiable if and only if there exists a structure that satisfies both the formula and the sentences of $\mathcal{T}$, or equivalently, if $\mathcal{T} \cup \{\psi\}$ is satisfiable [14]. Given a set $\Gamma \cup \{\psi\}$ of formulae over $\mathcal{T}$, we say that ψ is a $\mathcal{T}$-consequence of Γ, and write $\Gamma \models_{\mathcal{T}} \psi$, if and only if every model of $\mathcal{T} \cup \Gamma$ is also a model of ψ. Checking $\Gamma \models_{\mathcal{T}} \psi$ can be reduced in the usual way to checking the $\mathcal{T}$-satisfiability of $\Gamma \cup \{\neg\psi\}$.

The SMT-LIB initiative [9] aims at establishing a common standard for the specification of background theories, but most SMT solvers provide functions in addition to those specified in the SMT-LIB. Therefore, we describe here the fragments that we found in the SMT solvers Boolector, CVC3, and Z3 for the theory of linear, non-linear, and bit-vector arithmetic. We summarize the

1. Available at http://www.esbmc.org

syntax of these background theories as follows, using standard notations where appropriate:

$$\begin{aligned} F &::= F\ con\ F \mid \neg F \mid A \\ con &::= \wedge \mid \vee \mid \oplus \mid \Rightarrow \mid \Leftrightarrow \\ A &::= T\ rel\ T \mid Id \mid true \mid false \\ rel &::= < \mid \leq \mid > \mid \geq \mid = \mid \neq \\ T &::= T\ op\ T \mid \sim T \mid ite(F,\ T, T) \mid Const \mid Id \mid \\ &\quad Extract(T, i, j) \mid SignExt(T, k) \mid ZeroExt(T, k) \\ op &::= + \mid - \mid * \mid / \mid rem \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{|} \mid \oplus \mid @ \end{aligned}$$

Here, F denotes Boolean-valued expressions with atoms A, and T denotes terms built over integers, reals, and bit-vectors. The logical connectives con consist of conjunction ($\wedge$), disjunction ($\vee$), exclusive-or ($\oplus$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$). The bit-level operators are and (&), or (|), exclusive-or ($\oplus$), complement ($\sim$), right-shift (>>), and left-shift (<<). $Extract\,(T, i, j)$ denotes bit-vector extraction from bits i down to j to yield a new bit-vector of size $i-j+1$ while @ denotes the concatenation of the given bit-vectors. $SignExt\,(T, k)$ extends a bit-vector of size w to the signed equivalent bit-vector of size $w+k$, while $ZeroExt\,(T, k)$ extends the bit-vector with zeros to the unsigned equivalent bit-vector of size $w+k$. The conditional expression $ite(f, t_1, t_2)$ takes a Boolean formula f and depending on its value selects either the second or the third argument. The interpretation of the relational operators (i.e., $<$, $\leq$, $>$, $\geq$), the non-linear arithmetic operators $*$, $/$, remainder (*rem*) and the right-shift operator (>>) depends on whether their arguments are unsigned or signed bit-vectors, integers or real numbers. The arithmetic operators induce checks to ensure that the arithmetic operations do not overflow and/or underflow.

The array theories of SMT solvers are typically based on the McCarthy axioms [19]. The function *select(a, i)* denotes the value of a at index position i and *store(a, i, v)* denotes an array that is exactly the same as array a except that the value at index position i is v. Formally, the functions *select* and *store* can then be characterized by the following two axioms [16], [17], [18]:

$$\begin{aligned} &i = j \Rightarrow select\,(store\,(a, i, v)\,, j) = v \\ &i \neq j \Rightarrow select\,(store\,(a, i, v)\,, j) = select\,(a, j) \end{aligned}$$

Note that array bounds checks need to be encoded separately; the array theories employ the notion of unbounded arrays size, but arrays in software are typically of bounded size. Section 3 shows how to generate VCs to check for array bounds violation in programs. Equality on array elements is defined by the theory of equality with uninterpreted functions (i.e., $a = b \wedge i = j \Rightarrow select\,(a, i) = select\,(b, j)$) and the extensional theory of arrays then allows reasoning about array equality as follows [16], [17], [18]:

$$\begin{aligned} &a = b \Leftarrow \forall i \cdot select\,(a, i) = select\,(b, i) \\ &a \neq b \Rightarrow \exists i \cdot select\,(a, i) \neq select\,(b, i) \end{aligned}$$

Tuples are used to model the ANSI-C union and struct datatypes. They provide store and select operations similar to those in arrays, but work on the tuple elements. Each field of the tuple is represented by an integer constant. Hence, the expression $select(t, f)$ denotes the field f of tuple t while the expression $store(t, f, v)$ denotes a tuple t that at field f has the value v and all other fields remain the same.

In order to check the satisfiability of a formula, SMT solvers handle the terms in the given background theory using a decision procedure [22]. Pure SAT solvers, in contrast, require replacing all higher-level operators by bit-level circuit equivalents (also called *bit-blasting*), which destroys structural word-level information in the problem formulation and can cause scaling problems. For example, SAT solvers do not scale well when reasoning on the propositional encoding of arithmetic operators (e.g., multiplication), because the operands are treated as arrays of w (where w represents the bit width of the data type) unrelated propositional variables; consequently, computational effort can be wasted during the propositional satisfiability search [15]. However, SMT solvers are typically built on top of state-of-the-art SAT solvers and use bit-blasting as a last resort if the more abstract and less expensive techniques are not powerful enough to solve the problem at hand. For example, SMT solvers often integrate a simplifier, which applies standard algebraic reduction rules and contextual simplification.

3 SMT-BASED BMC FOR SOFTWARE

This section describes how we generate the VCs, in particular the encoding techniques that we use to convert the constraints and properties from the ANSI-C programs into the different background theories of the SMT solvers, and our approach to decide the best encoding and solver to be used during the verification process.

3.1 SMT-based BMC Formulation

In BMC, the program to be analyzed is modelled as a state transition system, which is extracted from the control-flow graph (CFG) [20]. This graph is built as part of a translation process from program text to single static assignment (SSA) form. A node in the CFG represents either a (non-) deterministic assignment or a conditional statement, while an edge in the CFG represents a possible change in the program's control location.

A state transition system $M = (S, T, S_0)$ is an abstract machine that consists of a set of states S, where $S_0 \subseteq S$ represents the set of initial states, and $T \subseteq S \times S$ is the transition relation, i.e., pairs of states specifying how the system can move from state to state. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to the *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in T$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a VC ψ such that ψ is satisfiable if and only if ϕ has a counter-example of length k or less. The VC ψ is a quantifier-free formula in a decidable subset of first-order logic, which is then checked for satisfiability by an SMT solver. In this work, we are interested in checking safety properties of single-threaded programs. The associated model checking problem is formulated by constructing the following logical formula:

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \qquad (1)$$

Here, ϕ is a safety property, I the set of initial states of M and $\gamma(s_j, s_{j+1})$ the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the executions of M of length i and (1) can be satisfied if and only if for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If (1) is satisfiable, then the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If (1) is unsatisfiable, we can conclude that no error state is reachable in k steps or less.

It is important to note that this approach can be used only to find violations of the property up to the bound k. In order to *prove* properties we need to compute the *completeness threshold* (*CT*), which can be smaller than or equal to the maximum number of loop-iterations occurring in the program [1], [4], [23], [24]. However, computing *CT* to stop the BMC procedure and to conclude that no counter-example can be found is as hard as model checking. Moreover, complex programs involve large data-paths and complex expressions. Consequently, even if we knew *CT*, the resulting formulae would quickly become too hard to solve and require too much memory to build. In practice we can thus only ensure that the property holds in M up to a given bound k. In our work, we focus on embedded software because it has characteristics that make it attractive for BMC, e.g., dynamic memory allocations and recursion are highly discouraged, and that make the limitations of *bounded* model checking less stringent.

3.2 Tool Architecture

Figure 1 shows the main software components of ESBMC. The white boxes (except for the SMT solver) represent the components that we reused from the CBMC model checker without any modification while the gray boxes with dashed lines represent the components that we modified in order to *(i)* generate VCs to check for memory leaks (implemented in *GOTO program*, see Subsection 3.5.7), *(ii)* to simplify the unrolled formula (implemented in *GOTO symex*, see Subsections 3.3 and 3.4) and *(iii)* to perform an up-front analysis in the CFG of the program to determine the best encoding and solver for a particular program (implemented in *GOTO symex*, see Subsection 3.5.8). The *GOTO program* component converts the ANSI-C program into a GOTO-program, which simplifies the representation (e.g., replacement of *switch* and *while* by *if* and *goto* statements), and handles the unrolling of the loops and the elimination of recursive functions. The *GOTO symex* component performs a symbolic simulation of the program. The gray boxes with solid lines represent new components that we implemented to encode the given constraints and properties of an ANSI-C program into a global logical context, using the background theories supported by the SMT solvers. We also implemented new components to interpret the counter-example generated by the supported SMT solvers. These software components must be implemented in the back-end to support each new SMT solver.

In the back-end of ESBMC, we build two sets of quantifier-free formulae C (for the constraints) and P (for the properties) so that C encodes the first part of ψ_k (more precisely, $I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$) and $\neg P$ encodes the second part (more precisely, $\bigvee_{i=0}^{k} \neg\phi(s_i)$). After that, we check $C \models_{\mathcal{T}} P$ using an SMT solver. If the answer is satisfiable, we have found a violation of the property ϕ, which is encoded in ψ_k. If not, the property holds up to the bound k.

3.3 Illustrative Example

We use the code shown in Figure 2 as a running example to illustrate the process of transforming a given ANSI-C program into SSA form and then into the quantifier-free formulae C and P shown in (2) and (3) respectively. This code implements a simplified version of the character stuffing technique, which avoids resynchronization after a transmission error by enclosing each frame with the ASCII character sequences DLE STX and DLE ETX [25]. Note that this syntactically valid ANSI-C program contains two subtle errors. In line 28 it writes to an address outside the allocated memory region of the array *out*. Additionally, the *assert* macro in line 29 fails when the ASCII character NUL is transmitted, i.e., when the condition of the *while* loop (line 11) does not hold. To detect this error, we use a non-deterministic input, i.e., we set the third position of array *in* (line 6) using $nd\_uchar()$, which can return any value in the range from zero to 255.

In reasoning about this C program, ESBMC checks 25 properties related to array bounds and overflow, and the user-specified assertion in line 29. ESBMC originally generates 63 VCs, but with the simplifications described in Section 3.4, only 9 remain. The first eight VCs check the bounds of the array *out* in lines 15, 18 and 28 and the last VC checks the user-specified assertion in line 29; note that the VCs to check the bounds of the array *out* are not simplified away due to the non-determinism in the array *in*, which does not allow checking statically

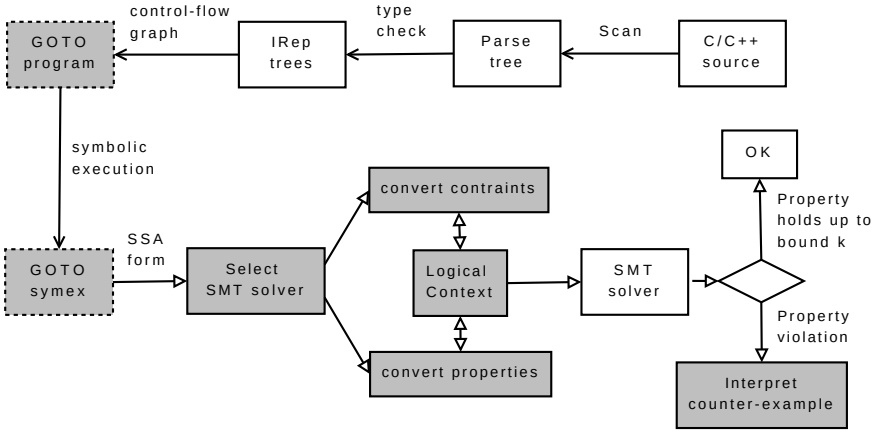

Fig. 1: Overview of the ESBMC architecture.

```
1  #define DLE 16
2  #define STX 2
3  #define ETX 3
4  uchar nd_uchar();
5  int main (void) {
6    uchar in[6] = {DLE, STX, nd_uchar(),
7                   DLE, ETX, '\0'};
8    uchar out[6];
9    int i = 0;
10   int j = 0;
11   while (in[i] != '\0') {
12     switch (in[i]) {
13       case (DLE):
14         if (in[i+1]==STX || in[i+1]==ETX) {
15           out[j] = in[i];
16         } else {
17           out[j] = in[i];
18           out[++j] = DLE;
19         };
20         break;
21       default:
22         out[j] = in[i];
23         break;
24     }
25     i++;
26     j++;
27   }
28   out[j] = '\0';
29   assert(out[4]==ETX || out[5]==ETX);
30   return 0;
31 }
```

Fig. 2: ANSI-C program with two violated properties.

whether the *if* statement in line 14 is *true* or *false*. In this particular example, CBMC v3.8 generates 136 VCs out of which 48 remain after simplification. The limited static analysis capability of CBMC thus leads to a substantially higher overhead in the solver.

However, before actually checking the properties, ESBMC unrolls the program using the simplification described in Section 3.4 and converts it into SSA form, as shown in Figure 3; note that the variable declarations as well as the return-statement are not shown. The SSA form only consists of conditional and unconditional assignments, where the left-hand side variable of each original assignment (e.g., $i = 0$), is replaced by a new variable (e.g., i_1), as well as assertions. removed. The SSA notation uses WITH as symbolic representation of the array *store* operator described in Section 2.2, i.e., a WITH $[i := v]$ is equivalent to *store(a, i, v)*.

$$C := \left[\begin{array}{l} in_1 = store(store(store(store(store(store(in_0, \\ 0, 16), 1, 2), 2, nd\_uchar_1), 3, 16), 4, 3), 5, 0) \\ \wedge\ i_1 = 0 \wedge j_1 = 0\ \wedge\ out_1 = store\,(out_0, 0, 16) \\ \wedge\ i_2 = 1 \wedge j_2 = 1\ \wedge\ out_2 = store\,(out_1, 1, 2) \\ \wedge\ g_1 = nd\_uchar_1 \neq 0 \\ \wedge\ g_2 = \neg\,(nd\_uchar_1 = 16) \\ \wedge\ out_3 = store\,(out_2, 2, nd\_uchar_1) \\ \wedge\ j_4 = 3 \\ \wedge\ \ldots \\ \wedge\ j_{10} = ite\,(\neg g_1, j_3, j_9) \\ \wedge\ out_{11} = store\,(out_{10}, j_{10}, 0) \end{array} \right] \tag{2}$$

$$P := \left[\begin{array}{l} j_5 \geq 0 \wedge j_5 < 6\ \wedge\ j_7 \geq 0 \wedge j_7 < 6 \\ \wedge\ j_8 \geq 0 \wedge j_8 < 6\ \wedge\ j_{10} \geq 0 \wedge j_{10} < 6 \\ \wedge\ (select\,(out_{11}, 4) = 3 \vee select\,(out_{11}, 5) = 3) \end{array} \right] \tag{3}$$

After this transformation, we build the constraints and properties as shown in formulae (2) and (3) using the background theories of the SMT solvers. Furthermore, we add Boolean variables (or *definition literals*) for each clause of the formula P in such a way that the definition literal is true iff a given clause of P is true. These definition literals are used to identify the VCs. In the example we add constraints as follows:

$$l_0 \Leftrightarrow j_5 \geq 0$$
$$l_1 \Leftrightarrow j_5 < 6$$
$$\ldots$$
$$l_9 \Leftrightarrow ((select\,(out, 4) = 3) \vee (select\,(out, 5) = 3))$$

and rewrite (3) as:

$$\neg P := \neg l_0 \vee \neg l_1 \vee \ldots \vee \neg l_9 \tag{4}$$

```
1  in1 == {16, 2, nd_uchar1, 16, 3, 0}
2  i1 == 0
3  j1 == 0
4  out1 == (out0 WITH [0:=16])
5  i2 == 1
6  j2 == 1
7  out2 == (out1 WITH [1:=2])
8  i3 == 2
9  j3 == 2
10 g1 == (nd_uchar1 != 0)
11 g2 == !(nd_uchar1 == 16)
12 out3 == (out2 WITH [2:=nd_uchar1])
13 j4 == 3
14 out4 == (out3 WITH [3:=16])
15 out5 == out2
16 j5 == j3
17 out6 == (out5 WITH [j5:=nd_uchar1])
18 out7 == (!g2 ? out4 : out6)
19 j6 == (!g2 ? j4 : j5)
20 i4 == 3
21 j7 == 1 + j6
22 out8 == (out7 WITH [j7:=16])
23 i5 == 4
24 j8 == 1 + j7
25 out9 == (out8 WITH [j8:=3])
26 i6 == 5
27 j9 == 1 + j8
28 out10 == (!g1 ? out2 : out9)
29 i7 == (!g1 ? i3 : i6)
30 j10 == (!g1 ? j3 : j9)
31 out11 == (out10 WITH [j10:=0])
```

Fig. 3: The program of Figure 2 in SSA form.

Note that the language-specific safety properties (e.g., out-of-bounds array indexing) and the user-specified properties that hold trivially in the code are already simplified away (e.g., by keeping track of the size of the array during the symbolic execution of the code). For instance, there is no need to generate VCs that check for array bounds violations on in, since i only takes the values from 0 to 4 when it is used in indexing the array, and the validity of the bounds checks can be evaluated statically.

We also simplify C and P by using local and recursive transformations in order to remove functionally redundant expressions and redundant literals as follows:

$$
\begin{array}{ll}
a \wedge \mathit{true} = a & a \wedge \mathit{false} = \mathit{false} \\
a \vee \mathit{false} = a & a \vee \mathit{true} = \mathit{true} \\
a \oplus \mathit{false} = a & a \oplus \mathit{true} = \neg a \\
\mathit{ite}\,(\mathit{true}, a, b) = a & \mathit{ite}\,(\mathit{false}, a, b) = b \\
\mathit{ite}\,(f, a, a) = a & \mathit{ite}\,(f, f \wedge a, b) = \mathit{ite}\,(f, a, b)
\end{array}
$$

Finally, the formula $C \wedge \neg P$ is passed to an SMT solver to check satisfiability. Our approach is slightly different from that of Armando et al. [3], who transform the ANSI-C code into conditional normal form as an intermediary step to encode C and P while we encode them directly from the SSA form.

3.4 Code Simplification and Reduction

We observed during development that constant propagation and forward substitution techniques [20] significantly improve the performance of ESBMC over a wide range of embedded software applications. We exploit the constant propagation technique to replace pointers to objects that are constants by the respective constant and to replace store operations that update the content of arrays, structs, and unions with constant values by these values.

```
1 ...
2 void puts(const char *s) {
3   while( *s ) {
4     putc(*s++);
5   }
6 }
7 ...
8 puts("blit: success");
```

Fig. 4: Code fragment of blit.

Figure 4 shows an example extracted from the Powerstone benchmark [37] to illustrate how constant propagation works for pointers in ESBMC. The function *puts* defined in line 2 is in line 8 called with a pointer to an array of constants, but CBMC's VCG still generates VCs to check for the bounds of the pointer s, as explained in Section 3.5.6. During the unrolling phase, we check whether the last value assigned to a pointer is a constant, and if so, we replace it by the constant and pass the modified expression to a simplifier, which is able to perform simple deductions before generating a VC to be encoded by the back-end.

We also propagate the *store*-operations for arrays, structs, and unions up to a certain level. Figure 5 shows an example extracted from the EUREKA benchmark [35] to illustrate how constant propagation works for arrays. In line 2, we initialize the first position of array a with a constant. In each iteration of the *for* loop, we add the value of the loop counter i to the last value written in array a and write the result to the next position of a (see line 4). After the loop, we check whether the assertion in line 6 holds. However, after unrolling the loop we obtain a VC involving a large expression $\mathit{store}(\ldots(\mathit{store}(\mathit{store}(\mathit{store}(a_0, 0, 1), 1, 2), 2, 4), 3, 7), \ldots)$ of nested store operations for a. Since all arguments except a_0 are constants, we can in principle check statically whether the assertion in line 6 holds. In practice, however, the model checker becomes slower than the SMT solvers in propagating these constants if the expressions become too large. In our benchmarks, we observed a substantial improvement in performance if we propagate the known constants up to six nested store operations (note that the value six was the optimum value that we obtained empirically with ESBMC using a large set of benchmarks). We thus reduce substantially the number of VCs, but we leave the harder cases for the SMT solvers.

```
1  ...
2  a[0]=1;
3  for( i=1; i<N; i++){
4    a[i]= a[i-1] + i;
5  }
6  assert(a[i-1]<2*1000);
7  ...
```

Fig. 5: Code fragment of SumArray.

We also observed that several applications repeat the same expression at many different places, especially after loop unrolling, in a way that the value of the operands does not change in between the occurrences. This can be detected easily in the SSA form and used for caching and forward substitution. Figure 6 shows a fragment of the Fast Fourier Transform (FFT) algorithm, extracted again from the SNU-RT benchmark [27], as an example of where the forward substitution technique can be applied. This occurs because the SSA representations of the two outermost *for* loops (in lines 6-15 and lines 8-14, respectively) will eventually contain several copies of the innermost *for* loop (lines 10-13), and thus the right-hand side of the assignment in line 11 is repeated several times in the SSA form, depending on the unwinding bound used to model check this program. Note that constant propagation means that the occurrence of i in line 11 is replaced by constant values. In the different copies of the unrolled outer loops we thus get multiple copies of the right-hand side that are identical in the SSA form.

```
1  typedef struct {
2    float real, imag;
3  } complex;
4  int n=1024;
5  complex x[1024], *xi;
6  for(le=n/2; le>0; le/=2) {
7    ...
8    for (j=0; j<le; j++) {
9      ...
10     for (i=j; i<n; i=i+2*le) {
11       xi = x + i;
12       ...
13     }
14   }
15 }
```

Fig. 6: Code fragment of Fast Fourier Transformation.

For example, if we set the unwinding bound k to 1024 (which is required because the upper bound n of the innermost *for*-loop is equal to 1024, see line 4), the *for*-loop in lines 6-15 will contain nine copies of the *for*-loop in lines 8-14, where the variable *le* will assume the values 512, 256, 128, ..., 1. In combination with constant propagation, the expression $x + i$ that is assigned to the pointer index *xi* is thus repeated up to nine times for each (propagated) value that i takes in the *for*-loop in lines 10-13. We thus include all expressions into a cache so that when a given expression is processed again in the program, we only retrieve it from the cache instead of creating a new copy using a new set of variables.

3.5 Encodings

This section describes the encodings that we use to convert the constraints and properties from the ANSI-C program into the background theories of the SMT solvers.

3.5.1 Scalar Data Types

We provide two approaches to model unsigned and signed integer data types, either as integers provided by the corresponding SMT-LIB theories or as bit-vectors of a particular bit width. For the encodings, we use the scalar datatypes supported by the front-end. The ANSI-C datatypes *int*, *long int*, *long long int*, and *char* are considered as *signedbv* with different bit widths (depending on the machine architecture) and the unsigned versions of these datatypes are considered as *unsignedbv*. We also support the C enumeration declaration (which is considered as *c_enum*) and we encode it as an integer type since the values of the enumeration are already generated by the front-end and obey normal scoping rules. For *double and float* we currently only support fixed-point arithmetic using *fixedbv*, but not full floating-point arithmetic; see the following section for more details.

The encoding of the relational (e.g., $<$, $\leq$, $>$, $\geq$) and arithmetic operators (e.g., $+$, $-$, $/$, $*$, *rem*) depends on the encoding of their operands. In the SAT-based version of CBMC [11], [30], the relational and arithmetic operators are transformed into propositional equations using a carry chain adder, and the size of their encoding thus depends on the size of the bit-vector representation of the scalar data types. In the SMT-based version of CBMC [6] only bit-vectors are used and the capabilities of the SMT solvers to model program variables through the corresponding numerical domains such as $\mathbb{Z}$ are not exploited, while the SMT-based BMC approach proposed by Armando et al. [3] does not support the encoding of fixed-point numbers.

We support all type casts, including conversion between integer and fixed-point types. These conversions are performed using the functions $Extract$, $SignExt$ and $ZeroExt$ described in Section 2.2. Similarly, upon dereferencing, the object that a pointer points to is converted using the same word-level functions. The datatype *bool* is converted to *signedbv* and *unsignedbv* using ite. In addition, *signedbv* and *unsignedbv* variables are converted to *bool* by comparing them to constants whose values represent zero in the corresponding types.

3.5.2 Fixed-Point Arithmetic

Embedded applications from domains such as discrete control and telecommunications often require arithmetic over non-integral numbers. However, an encoding of the full floating-point arithmetic into the BMC framework

leads to large formulae; instead, we over-approximate it by fixed-point arithmetic, which might introduce behaviour that is not present in a real implementation. We use two different representations to encode non-integral numbers, binary (when dealing with bit-vector arithmetic) and decimal (when dealing with rational arithmetic). In this way, we can explore the different background theories of the SMT solvers and trade off speed and accuracy as further described in Section 3.5.8.

We encode fixed-point numbers using the integral and fractional parts separately [28]. Given a rational number that consists of an integral part I with m bits and a fractional part F with n bits, we represent it by $\langle I.F \rangle$ and interpret it as $I + F/2^n$. For instance, the number 0.75 can be represented as $\langle 0000.11 \rangle$ in base 2 while 0.125 can represented as $\langle 0000.001 \rangle$.

We encode fixed-point arithmetic using bit-vector arithmetic as in the binary encoding, but we assume that the operands have the same bitwidths both before and after the radix point. If this is not the case, we pad the shorter bit sequence and add zeros from the right (if there are bits missing in the fractional part), e.g., 0.75 + 0.125 = $\langle 0000.1100 \rangle$ + $\langle 0000.0010 \rangle$) or from the left (if there are bits missing before the radix point).

We encode fixed-point arithmetic using rational arithmetic by rounding the fixed-point numbers to rationals in base 10. We extract the integral and fractional parts and convert them to integers I and F, respectively; we then divide F by 2^n, round the result to a given number of decimal places, and convert everything to a rational number in base 10. For example, with $m = 2$, $n = 16$, and six places decimal precision, the number 3.9 (11.1110011001100110) is converted to $I = 3$, and $F = 58982/2^{16}$, and finally to $3899994/100000$. As a result, the arithmetic operations are performed in the domain of $\mathbb{Q}$ instead of $\mathbb{R}$ and there is no need to add missing bits to the integer and fractional parts.

In general, the drawback is that some numbers are not precisely represented with fixed-point arithmetic. As an example, if $m = 4$ and n=4, then the closest numbers to 0.7 are 0.6875 ($\langle 0000.1011 \rangle$) and 0.75 ($\langle 0000.1100 \rangle$). As a result, the number needs to be rounded and the deviation might eventually change the control flow of the program.

3.5.3 Arithmetic Overflow and Underflow

Arithmetic overflow and underflow are frequent sources of bugs in embedded software. ANSI-C, like most programming languages, provides basic data types that have a bounded range defined by the number of bits allocated to them. Some model checkers treat program variables either as unbounded integers (e.g., Blast [62]) or do not generate VCs related to arithmetic overflow (e.g., SMT-CBMC [3], SMT-based CBMC [6] and F-Soft [4]), and can consequently produce false positive results. In our work, we generate VCs related to arithmetic overflow and underflow following the ANSI-C standard. This requires that, on arithmetic overflow of *unsigned* integer types (e.g., *unsigned int*), the result must be interpreted using modular arithmetic as $r \ mod \ 2^w$, where r is the expression that caused overflow and w is the bit-width of the result type [5]. Hence, in this encoding the result of the expression is one greater than the largest value that can be represented by the result type. This semantics can be encoded trivially using the background theories of the SMT solvers. For each unsigned integer expression, we generate a literal $l_{unsigned\_overflow}$ to represent the validity of the unsigned operation and add the constraint:

$$l_{unsigned\_overflow} \Leftrightarrow (r - (r \ mod \, 2^w)) < 2^w$$

The ANSI-C standard does not define any behaviour on arithmetic overflow of *signed* types (e.g., *int*, *long int*), and only states that integer *division-by-zero* must be detected. In addition to *division-by-zero* detection, we consider arithmetic overflow of *signed* types on addition, subtraction, multiplication, division and negation operations by defining boundary conditions. For example, we define a literal $l_{overflow^*_{x,y}}$ that is true iff the multiplication of x and y exceeds LONG_MAX (i.e., $x * y > LONG\_MAX$) and another literal $l_{underflow^*_{x,y}}$ that is true iff the multiplication of x and y is below LONG_MIN. We use a literal $l_{res\_op^*}$ to denote the validity of the signed multiplication with the following constraint:

$$l_{res\_op^*} \Leftrightarrow (\neg l_{overflow^*_{x,y}} \wedge \neg l_{underflow^*_{x,y}})$$

The overflow and underflow checks on the remaining operators are encoded in a similar way.

3.5.4 Arrays

Arrays are encoded in a straight-forward manner using the SMT domain theories, since the *WITH* operator and index operator $[]$ can be mapped directly to the functions *store* and *select* of the array theory presented in Section 2.2 [11], [29]. For example, the assignment a' = $a \ WITH \ [i := v]$ is encoded with a store operation a' = $store\,(a, i, v)$ while $x = a[i]$ is encoded with a select operation x = $select\,(a, i)$. In the array theory, the arrays are unbounded, but in a program, if an array index is out of bounds, the value selected is undefined and an out-of-bounds write can cause a crash. In order to check for array bounds violations, we simply keep track of the size of the array and generate VCs that are provable iff the indexing expression is within the array's bounds.

```
1 int i, a[N];
2 ...
3 for(i=0; i<N; i++)
4   a[i+1]=2*i;
5 ...
```

Fig. 7: Array out of bounds example.

As an example, consider the code fragment shown in Figure 7. In order to check for the array bounds in line

4 of Figure 7, we create a VC to check the array index i only for the last iteration of the *for*-loop since for all i with $i < N - 1$ we can statically infer that there is no array bounds violation. This VC does not require the array theory and can be written as follows:

$$i < N \Rightarrow (i + 1 < N) \tag{5}$$

Armando et al. [3] also encode programs with arrays using the array theory of the SMT solvers, but they do not generate VCs to check for array bounds violation. The SAT-based version of CBMC generates such VCs but the underlying array representation is fundamentally different. Each array a of size s is replaced by s different scalar variables $a_0, a_2, \ldots, a_{s-1}$ and $a' = store(a, i, v)$ is then represented by the following formula [11], [30]:

$$\bigwedge_{j=0}^{s-1} a'_j = ((i = j) \wedge v) \vee (\neg (i = j) \wedge a_j) \tag{6}$$

Similarly, $b = select(a, i)$ is represented as follows:

$$\bigwedge_{j=0}^{s-1} (i = j) \Rightarrow (b = a_j) \tag{7}$$

The size of the propositional formulae (6) and (7) depends on the bit-width of the scalar data types and the size of the arrays occurring in the program, as observed by [3]. In addition, all high-level structure present in the original formula is lost. In contrast, our approach yields more compact VCs and keeps the inherent structure.

3.5.5 Structures and Unions

Structures and unions are encoded using the theory of tuples in SMT and we map update and access operations to the functions *store* and *select* of the theory of tuples presented in Section 2.2. Let w be a structure type, f be a field name of this structure, and v be an expression matching the type of *f*. The expression *store(w,f,v)* returns a tuple that is exactly the same as w except that the value of field f is v; all other tuple elements remain the same. Formally, if *w'=store(w,f,v)* and j is a field name of w, then:

$$w'.j = \begin{cases} v & \text{if } j = f, \\ w.j & \text{if } j \neq f \end{cases} \tag{8}$$

We encode unions in a similar way. The difference is that we add an additional field l to indicate the number of the field that was used last for writing into the union. This is used to insert the required type-cast operations if any subsequent read access uses a different field.

In contrast, the SMT-based BMC approach proposed by Armando et al. [3] does not support unions; Clarke [11], [30] and Kroening [6] encode structs and unions by concatenating and extracting the fields. This approach, however, might be less scalable because high-level information is lost and therefore, needs to be re-discovered by the SAT or SMT solver (possibly with a substantial performance penalty).

3.5.6 Pointers

In ANSI-C, pointers (and pointer arithmetics) are used as alternative to array indexing: $*(p + i)$ is equal to $a[i]$, if p has been assigned a (see Figure 8). The front-end of CBMC removes all pointer dereferences during the unwinding phase and treats pointers as program variables. CBMC's VCG uses the predicate *SAME_OBJECT* to represent that two pointer expressions point to the same memory location or same object. Note that *SAME_OBJECT* is not a safety property, but is mainly used to produce sensible error messages. The VCG generates safety properties that check that *(i)* the pointer offset does not exceed the object bounds (represented by *LOWER_BOUND* and *UPPER_BOUND*) and *(ii)* the pointer is neither `NULL` nor an invalid object (represented by *INVALID_POINTER*). Our approach is similar to the encoding of CBMC into propositional logic, but we use the background theories such as tuples, integer and bit-vector arithmetic while CBMC encodes them by concatenating and extracting the bit-vectors, which operates at the bit-level and does not exploit the structure provided by the higher abstraction levels and is thus less scalable.

We encode pointers using two fields of a tuple *p* such that *p.o* encodes the object the pointer points to, while the *p.i* encodes an offset within that object. Note that the object can be an array, a struct, or a scalar and that the interpretation of *p.i* depends on the type of the object: for arrays, it denotes the index, for structs the field, and for scalar it is fixed to zero. Note further that we update the object field *p.o* dynamically (using the *store* operation of the tuple theory) to accommodate changes of the object that the pointer points to.

Formally, let p_a and p_b be pointer variables pointing to the objects a and b. We encode *SAME_OBJECT* by a literal l_{same_object} with the following constraint:

$$l_{same\_object} \Leftrightarrow (p_a.o = p_b.o) \tag{9}$$

A pointer p may point to a set of objects during its lifetime. We thus check the *SAME_OBJECT* property whenever we check the value pointed by pointer p or whether the offset of p is within the bounds of an object a. This means that we generate l_{same_object} for each expression that uses p as array indexing. Formally, in order to check the pointer index, we define the upper and lower bound of an object b by b_u and b_l respectively. We then encode the properties *LOWER_BOUND* and *UPPER_BOUND* by creating two literals l_{lower_bound} and l_{upper_bound} with the following constraints:

$$\begin{aligned} l_{lower\_bound} &\Leftrightarrow \neg (p_a.i < b_l) \vee \neg (p_a = p_b) \\ l_{upper\_bound} &\Leftrightarrow \neg (p_a.i \geq b_u) \vee \neg (p_a = p_b) \end{aligned} \tag{10}$$

To check invalid pointers, the `NULL` pointer is encoded as a unique identifier denoted by η and an invalid object is denoted by ν. If p denotes a pointer expression, we encode the property *INVALID_POINTER* by a literal

$l_{invalid\_pointer}$ with the following constraint:

$$l_{invalid\_pointer} \Leftrightarrow (p.o \neq \nu) \wedge (p.i \neq \eta) \tag{11}$$

As example, consider the C program of Figure 8 where the pointer p points to the array a as shown in line 3. We build the constraints and properties shown in (12) and (13) so that the assignment *p=a* in line 3 is converted into a tuple p. The second and third conjuncts $p_1 = store\,(p_0, 0, a)$ and $p_2 = store\,(p_1, 1, 0)$ of (12) store the object (i.e., array a) and the index 0 at the first two positions of the tuple p.

$$C := \left[\begin{array}{l} i_0 = 0 \wedge\ p_1 = store\,(p_0, 0, a) \\ \wedge\ p_2 = store\,(p_1, 1, 0) \wedge\ g_1 = (x_1 = 0) \\ \wedge\ a_1 = store(a_0, i_0, 0) \\ \wedge\ a_2 = a_0 \\ \wedge\ a_3 = store(a_2, 1 + i_0, 1) \\ \wedge\ a_4 = ite(g_1, a_1, a_3) \\ \wedge\ p_3 = store\,(p_2, 1, select\,(p_2, 1) + 2) \end{array} \right] \tag{12}$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\ 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\ select\,(p_3, select\,(p_3, 1)) = a \\ \wedge\ select\,(select\,(p_4, 0)\,, select\,(p_4, 1)) = 1) \end{array} \right] \tag{13}$$

In order to check the property specified in line 8, we first add the value 2 to *p.i* (i.e., $p_3 = store\,(p_2, 1, select\,(p_2, 1) + 2)$ shown in the last expression of (12)) and then check whether p and a point to the same memory location (as shown in the next to last expression of (13)). As *p.i* exceeds the size of the object stored in *p.o*, (i.e., array *a*), then the *SAME_OBJECT* property is "violated" and thus the `assert` macro in line 8 fails because $a[2]$ is unconstrained (i.e., it is a free variable as described in Section 3.5.4).

```
1 int main() {
2   int a[2], x, i=0, *p;
3   p=a;
4   if (x==0)
5     a[i]=0;
6   else
7     a[i+1]=1;
8   assert(*(p+2)==1);
9 }
```

Fig. 8: C program with pointer to an array.

Structures consisting of n fields with scalar data types are also manipulated like an array with n elements. This means that the front-end of CBMC allows us to encode the structures by using the usual update and access operations. If the structure contains arrays, pointers and scalar data types, then *p.i* points to the object within the structure only. As an example, Figure 9 shows a C program that contains a pointer to a *struct* consisting of two fields (an array *a* of integer and a *char* variable *b*). As the *struct y* is declared as global in Figure 9 (see lines 1-4), its members must be initialized before performing any operation [5], as shown in the first two lines of (14). The assignment $p = \&y$ (see line 7 of Figure 9) is encoded by assigning the structure y to the field $p_1.o$ and the value 0 to the field $p_1.i$.

```
1 struct x {
2   int a[2];
3   char b;
4 } y;
5 int main(void) {
6   struct x *p;
7   p=&y;
8   p->a[1]=1;
9   p->b='c';
10  assert(p->a[1]==1);
11  assert(p->b=='c'); //ASCII 99
12 }
```

Fig. 9: C program with pointer to a struct.

$$C := \left[\begin{array}{l} y_0.b := 0 \\ \wedge\ y_1 := store(store(y_0.a, 0, 0), 1, 0) \\ \wedge\ p_1.o := y\ \wedge\ p_1.i := 0 \\ \wedge\ y_2 := store(y_1, a, store(y_1.a, 1, 1)) \\ \wedge\ y_3 := store(y_2, b, 99) \end{array} \right] \tag{14}$$

$$P := \left[\begin{array}{l} select(select(y_3, a), 1) = 1 \\ \wedge\ select(y_3, b) = 99 \end{array} \right] \tag{15}$$

3.5.7 Dynamic Memory Allocation

Although dynamic memory allocation is discouraged in embedded software, ESBMC is capable of model checking programs that use it through the ANSI-C functions *malloc* and *free*. We model memory just as an array of bytes and exploit the arrays theories of SMT solvers to model read and write operations to the memory array on the logic level. ESBMC checks three properties related to dynamic memory allocation; in particular, it checks whether *(i)* the argument to any *malloc*, *free*, or dereferencing operation is a dynamic object (*IS_DYNAMIC_OBJECT*), *(ii)* the argument to any *free* or dereferencing operation is still a valid object (*VALID_OBJECT*), and *(iii)* whether the memory allocated by the *malloc* function is deallocated at the end of an execution (*DEALLOCATED_OBJECT*) [31]. The last check extends the CBMC's VCG.

Formally, let p_o be a pointer expression that points to the object o of type t and let m be a memory array of type t and size n, where n represents the number of elements to be allocated. In our encoding, the representation of each dynamic object d_o contains a unique identifier ρ that indicates the objects "serial number" in the sequential order of all dynamically allocated objects (i.e., $0 \leq \rho < k$, where k represents the total number of dynamic objects).

Each dynamic object consists of the memory array m, the size in bytes of m, the unique identifier ρ. and the location in the execution where m is allocated, which is used for error reporting.

To detect invalid reads/writes, we check whether d_o is a dynamic object and also whether p_o is within the bounds of the memory array. Let i be an integer variable that indicates the position in which the object pointed to by p_o must be stored in the memory array m. We encode *IS_DYNAMIC_OBJECT* as a literal $l_{is\_dynamic\_object}$ with the following constraint:

$$l_{is\_dynamic\_object} \Leftrightarrow \left(\bigvee_{n=0}^{k-1} d_o.\rho_j = n \right) \wedge (0 \leq i < n) \quad (16)$$

To check for invalid objects, we add one additional bit field ν to each dynamic object to indicate whether it is still alive or not. We set ν to *true* when the function *malloc* is called to denote that the object is alive. When the function *free* is called, we set ν to *false* to denote that the object is no longer alive. We then encode *VALID_OBJECT* as a literal l_{valid_object} with the following constraint:

$$l_{valid\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow d_o.\nu) \quad (17)$$

To detect forgotten memory, we check, at the end of the (unrolled) program, for each dynamic object whether it has been deallocated by the function *free*. We can thus use the existing flag, encoding *DEALLOCATED_OBJECT* as a literal $l_{deallocated\_object}$ with the following constraint:

$$l_{deallocated\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow \neg d_o.\nu) \quad (18)$$

Note that the difference between *VALID_OBJECT* and *DEALLOCATED_OBJECT* is the location at which they are checked: *VALID_OBJECT* is checked for each access to a pointer variable, while *DEALLOCATED_OBJECT* is checked only immediately before the (unrolled) program terminates. Note further that both allocation location and size of each dynamic object are immutable whereas the bit field ν is updated when the functions *malloc* and *free* are called.

3.5.8 Exploiting Representative Datatypes

Modern SMT solvers provide ways to model the program variables either as bit-vectors or as elements of an abstract numerical domain (e.g., $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$). If the program variables are modelled as bit-vectors of a fixed size, then the result of the analysis can be precise (w.r.t. the ANSI-C semantics), depending on the size considered for the bit-vectors. In contrast, if the program variables are modelled using the abstract numerical domains, then the result of the analysis is independent from the actual binary representation, but it may not be precise when arithmetic expressions are involved. As example consider the following small C program from [11], [30] as shown in Figure 10.

This program nondeterministically selects two values of type *unsigned char* and uses bitwise AND, right- and left-shift operations to multiply them. Reasoning about this program by means of integer arithmetic produces wrong results if the bit-level operators are treated as uninterpreted functions (UFs). Although UFs simplify the proofs, they ignore the semantics of the operators and consequently make the formula weaker. In addition, the majority of the software model checkers (e.g., SMT-CBMC [3] and BLAST [62]) fail to check the assertion in line 9. On the other hand, bit-vector arithmetic allows us to encode bit-level operators in a more accurate way. However, in our benchmarks, we noted that the majority of VCs are solved faster if we model the basic datatypes as $\mathbb{Z}$ and $\mathbb{R}$. Consequently, we have to trade off between *speed* and *accuracy* which are two competing goals in formal verification of software using SMT.

```
1  int main() {
2    unsigned char a, b;
3    unsigned int result=0, i;
4    a=nd_uchar();
5    b=nd_uchar();
6    for(i=0; i<8; i++)
7      if((b>>i)&1)
8        result+=(a<<i);
9    assert(result==a*b);
10 }
```

Fig. 10: A C program that uses shift-and-add to multiply two numbers.

Based on the extent to which the SMT solvers support the domain theories and on experimental results obtained with a large set of benchmarks, we developed a simple but effective heuristic to determine the best representation for the program variables as well as the best SMT solver to be used in order to check the properties of a given ANSI-C program. Our default representation for encoding the constraints and properties of ANSI-C programs are integers and reals, respectively, and our default solver is Z3. We then explore the CFG representation of the program. If we find expressions that involve bit operations (e.g., <<, >>, &, |, $\oplus$) or typecasts from signed to unsigned datatypes and vice-versa, we encode the corresponding variables as bit-vectors and either switch the SMT solver to Boolector (if no pointers are used) or we keep Z3 (if pointers are used). We adopted this strategy because we are able to implement the theory of tuples on top of Z3 to model pointers and thus exploit the structure provided by the word-level instead of bit-level models (i.e., instead of concatenating and extracting bit-vectors, which is the approach used by CBMC [11], [30] and has not shown success in practice due to the loss of structure associated with the translation process) [32].

4 Experimental Evaluation

The experimental evaluation of our work consists of five parts. After describing the setup in Section 4.1, we compare in Section 4.2 the SMT solvers Boolector, CVC3, and Z3 to identify the most suitable SMT solver for

further development and experiments. In Section 4.3, we evaluate the simplification techniques proposed in Section 3.4. In Section 4.4 we check the error detection capability of ESBMC over a large set of both correct and buggy ANSI-C programs. In the last two sections, we evaluate ESBMC's performance relative to that of two other ANSI-C BMC tools. In Section 4.5, we compare ESBMC and SMT-CBMC, using SMT-CBMC's own benchmark suite, while we compare ESBMC and CBMC in the final Section 4.6, using a variety of programs, including embedded software used in telecommunications, control systems, and medical devices.

4.1 Experimental Setup

We used benchmarks from a variety of sources to evaluate ESBMC's precision and performance, which include embedded systems benchmark suites and applications as well as other testsuites and applications, including the SAT solver PicoSAT [42], the open-source applications *flex* [43] and *git-remote* [44], and a flasher manager application [45]. We also extracted one particular application from the CBMC manual [11] that implements the multiplication of two numbers using bit-level operations.

The PowerStone [37] suite contains graphics applications, paging communication protocols and bit shifting applications. The SNU-RT [27] suite consists of matrix and signal processing functions such as matrix multiplication and decomposition, quadratic equations solving, cyclic redundancy check, fast fourier transform, LMS adaptive signal enhancement, and JPEG encoding. We use the non-deterministic version of these benchmarks where all inputs are replaced by non-deterministic values. We also a cubic equation solver from the MiBench [41] suite. The HLS suite [33] contains programs that implement the encoder and decoder of the adaptive differential pulse code modulation (ADPCM).

The NXP [40] benchmarks are taken from the set-top box of NXP Semiconductors that is used in high definition internet protocol and hybrid digital TV applications. The embedded software of this platform relies on the Linux operating system and makes use of different applications such as *(i) LinuxDVB* that is responsible for controlling the front-end, tuners and multiplexers, *(ii) DirectFB* that provides graphics applications and input device handling and *(iii) ALSA* that is used to control the audio applications.

The NECLA [36] and VERISEC [34] benchmarks are not specifically related to embedded software, but they allow us to check ESBMC's error-detection capability easily since they provide ANSI-C programs with and without known bugs. Here, we use the suffix "-bad" to denote the subset with seeded errors, and "-ok" to denote the supposedly correct ("golden") versions. The programs make use of dynamic memory allocation, interprocedural dataflow, aliasing, pointers typecast and string manipulation. In addition, we used some programs from the well-known Siemens [39] test suite, including pattern matching and string processing, statistics, and aerospace applications. The EUREKA [3] benchmarks finally contain programs that allow us to assess the scalability of the model checking tools on problems of increasing complexity [3].

All experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 4 GB of RAM running Linux OS. For all benchmarks, the time limit has been set to 3600 seconds for each individual property. All times given are wall clock time in seconds as measured by the unix *time* command.

4.2 Comparison of SMT solvers

As a first step, we compared to which extent the SMT solvers support the domain theories that are required for SMT-based BMC of ANSI-C programs. For this purpose, we analyzed the SMT solvers Boolector (V1.4), CVC3 (V2.2), and Z3 (V2.11). In the *theory of linear and non-linear arithmetic*, CVC3 and Z3 do not support the remainder operator, but they allow us to use axioms to define it. Currently, Boolector does not support the theory of linear and non-linear arithmetic at all. In the *theory of bit-vectors*, CVC3 does not support the division and remainder operators for bit-vectors representing signed and unsigned integers. However, in all cases, axioms can be used in order to define the missing operators. Boolector and Z3 support all word-level, bit-level, relational, arithmetic functions over unsigned and signed bit-vectors. In the theories of *arrays* and *tuples*, the verification problems only involve selecting and storing elements from/into arrays and tuples, respectively, and both domains thus comprise only two operations. These operations are fully supported by CVC3 and Z3; Boolector supports only the theory of arrays but not that of tuples.

We then used 15 ANSI-C programs to compare the performance of Boolector, CVC3, and Z3 as ESBMC backends. The programs 1-8 allow us to assess the scalability of the model checking tools on problems of increasing complexity [3] and the programs 9-15 contain typical ANSI-C constructs found in embedded software, i.e., they contain linear and non-linear arithmetic and make heavy use of bit operations.

Table 1 shows the results of the comparison. Here, L is the number of lines of code, B the unwinding bound, and P the number of properties verified, for each ANSI-C program. We checked for language-specific safety properties (e.g., pointer safety, array bounds, division by zero) as well as user-specified properties. For each solver, we provide the total time (in seconds) to check all properties of each program at the same time, using the specified unwinding bound, as well as the solver time itself. The difference between both times is spent in the ESBMC front-end. In addition, we provide (in brackets) the timings using the SMT-LIB interface instead of the native API of the solver. The fastest time for each program is shown in bold. We also indicate whether ESBMC fails during the verification process, either due

	Program	L	B	P	CVC3 (v2.2)		Boolector (v1.4)		Z3 (v2.11)	
					Solver	Total	Solver	Total	Solver	Total
1	EUREKA.BubbleSort	43	35	17	14 (3)	17 (5)	<1 (<1)	**2 (2)**	<1 (<1)	**2** (3)
		43	70	17	M_b (16)	M_b (33)	3 (1)	**16** (17)	3 (1)	**16** (17)
		43	140	17	M_b (M_b)	M_b (M_b)	85 (53)	282 (311)	65 (11)	**265** (269)
2	EUREKA.SelectionSort	34	35	17	17 (2)	18 (3)	<1 (<1)	**1 (1)**	<1 (<1)	**1 (1)**
		34	70	17	M_b (8)	M_b (17)	1 (<1)	**9** (10)	1 (1)	**9** (11)
		34	140	17	M_b (42)	M_b (209)	10 (3)	**161** (171)	12 (6)	165 (173)
3	EUREKA.InsertionSort	34	35	17	2 (3)	4 (5)	<1 (<1)	**3 (3)**	<1 (<1)	**3 (3)**
		34	70	17	3 (11)	14 (24)	4 (<1)	15 (13)	2 (1)	**12** (14)
		34	140	17	21 (67)	**194** (283)	193 (3)	350 **(219)**	42 (7)	212 (222)
4	EUREKA.BellmanFord	49	20	33	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
5	EUREKA.Prim	79	8	30	<1 (1)	5 (2)	<1 (<1)	**<1 (<1)**	<1 (<1)	**<1 (<1)**
6	EUREKA.StrCmp	14	1000	6	4 (444)	**11** (454)	192 (248)	195 (257)	32 (37)	35 (46)
7	EUREKA.SumArray	12	1000	7	<1 (106)	**1** (107)	<1 (<1)	**1 (1)**	9 (<1)	10 **(1)**
8	EUREKA.MinMax	19	1000	9	T_b (M_b)	T_b (M_b)	38 (2)	42 (7)	2 (1)	**6** (7)
9	SNU-RT.Fibonacci	40	30	4	<1 (<1)	39 **(38)**	<1 (<1)	39 **(38)**	<1 (<1)	39 **(38)**
10	SNU-RT.bs	95	15	7	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
11	SNU-RT.lms	258	202	23	97 (17)	**225** (324)	<1 (<1)	303 (307)	3 (<1)	306 (307)
12	MiBench.Cubic	66	5	5	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
13	CBMC.BitWise	18	8	1	3 (6)	**3** (6)	7 (8)	7 (8)	30 (26)	30 (26)
14	HLS.adpcm_encode	149	200	12	<1 (21)	**6** (26)	<1 (<1)	**6 (6)**	<1 (<1)	**6 (6)**
15	HLS.adpcm_decode	111	200	10	<1 (24)	**3** (27)	<1 (<1)	**3 (3)**	<1 (<1)	**3 (3)**

TABLE 1: Results of the comparison between CVC3, Boolector and Z3. Time-outs are represented with T in the Time column; Examples that exceed available memory are represented with M in the Time column.

to a time out (T) or due to memory overflow (M). All failures occurred in the back-end (i.e., solver), which is indicated by the subscript b.

As we can see in Table 1, if we use the native API of the solvers, Z3 usually runs slightly faster than Boolector and CVC3; however, both CVC3 and Boolector are faster for some programs. Generally the difference between the solvers (in particular between Boolector and Z3) are small, although CVC3 fails for some examples. If we use the SMT-LIB interface, the situation changes, and Boolector runs slightly faster than Z3 and CVC3. However, similar to case of the native API, it is not always the fastest solver; again, the differences are generally small, and even smaller than when using the native API.

Generally, the native API is slightly faster than the SMT-LIB interface, although the difference is small. However, there are a few notable exceptions. Using the SMT-LIB interface, CVC3 scales better for *BubbleSort* and *SelectionSort*, but slows down substantially for *StrCmp* and *SumArray*. We manually inspected the respective VCs and found that their structure is essentially the same. We conclude that the SMT-LIB interface of CVC3 lacks some optimization during the preprocessing. Similarly, Boolector speeds up for *InsertionSort* using the SMT-LIB API, but the structure of the VCs using both APIs is also the same; similarly, we conclude that the SMT-LIB interface enables some optimization during the preprocessing.

We decided to continue the development with Z3 and Boolector using both the native and SMT-LIB APIs since CVC3 does not scale so well and fails to check the three benchmarks *BubbleSort*, *SelectionSort* and *MinMax*.

4.3 Performance Improvement

We evaluate the effectiveness of the simplification techniques described in Section 3.4 using 174 programs, with a total size of 70K lines of code, taken from the benchmark suites Siemens, SNU-RT, PowerStone, NECLA and NXP. With all optimizations enabled, ESBMC can check all 174 programs in 439 seconds, which serves as our baseline. We then evaluate the effect of the simplifications by disabling them one at a time as follows: constant propagation of store operations for arrays, structs and unions (CP_{store}); constant propagation for constant strings (CP_{String}); forward substitution (FS); and removal of functionally redundant literals and variables ($FRLV$). We set the time out to 180 seconds because it is longest time to check a given program with all optimizations enabled.

Surprisingly, ESBMC performs better when we disable the removal of functionally redundant literals and variables ($FRLV$) and checks all 174 programs in 423 seconds. We can thus conclude that the SMT solvers already eliminate the functionally redundant literals and variables during the preprocessing phase in a more efficient way. Fortunately, all other simplifications pay off. Using CP_{store}, ESBMC checks 170 programs in 1059 seconds and times out in four programs. With CP_{String}, it checks 173 programs in 590s and times out in one

program, and with FS, it checks 171 programs in 972 seconds and times out in three programs. The optimizations are complementary in the sense that disabling each one of them causes ESBMC to time out on different programs. Moreover, their effect is not only restricted to the programs that ESBMC fails to check when they are disabled: on the remaining 166 programs, disabling CP_{store} causes an average slow-down of more than 30%, although the effects are less pronounced, or even reversed, for disabling CP_{String} and FS, with a slow-down of approx. 8%, and a speed-up of approx. 4%, respectively.

4.4 Error-Detection Capability

As a third step, we analyze to which extent ESBMC is able to handle and detect errors in standard ANSI-C benchmarks. Table 2 summarizes the results. Here, N is the number of programs in the benchmark suite, while ΣL and ΣP give its total size (in lines of code) and the total number of claims checked, respectively. The table again shows both the solver and total verification time. In the last three columns, N_e is the number of programs in which ESBMC has detected violations of safety properties and user-specified assertions, "true" reports the number of property violations that correspond to true, confirmed faults, "false" reports the number of false negatives produced by ESBMC.

The EUREKA suite only contains correct programs. However, in the NECLA and VERISEC suites, ESBMC is able to detect errors related to buffer overflow, aliasing, dynamic memory allocation, and string manipulation; in particular, it detects all seeded errors in the versions NECLA-bad and VERISEC-bad. Moreover, ESBMC could verify two programs that were originally in NECLA-bad but did not contain any seeded errors; the benchmark creators confirmed that these programs were misclassified and subsequently changed the error seeding.

Surprisingly, ESBMC also detects errors in the supposedly correct golden versions. In NECLA-ok, ESBMC finds three property violations in two programs, which have been confirmed as true faults by the benchmark creators [46]. The first is an array bounds violation, caused by an indexing expression $x\%32$ that can become negative for negative inputs x. The other two are also related to array bounds violations, but are caused by repeated in-place updates of a buffer using the *strcat*-function, which also appends a new NULL-character at the end of the new string formed by the concatenation of both arguments; this NULL-character then causes the violation in the last iteration of the loop. In VERISEC-ok, ESBMC finds 15 property violations in nine programs, which have also been confirmed by the benchmark creators [47]. All violations are related to arithmetic overflow on the typecast operation caused by assignments of the form *c=i*, where *c* is declared as a *char* and *i* as an *int*.

In the WCET test suite, ESBMC finds four property violations in two programs, which we inspected manually. Two violations point to possible overflows that stem from assignments between incompatible datatypes (e.g., *long int* vs. *int*), which are indeed errors; a further violation points to a potential division by zero error, which is unlikely to be uncovered by testing, as it requires an entire array to be randomly initialized with zeroes. The final property indicates an arithmetic overflow in an expression *StopTime-StartTime*, but this is a false negative, since both variables are guaranteed to be positive at runtime, and moreover, *StopTime* is always larger than *StartTime*. This false negative can be suppressed by adding an assumption on the return values to the *ttime*-function that is used to compute both variables. Finally, ESBMC finds array bounds violations and overflows in arithmetic expressions in four of the SNU-RT benchmarks and invalid pointers in one of the PowerStone benchmarks; we confirmed by inspection that these are indeed faults.

4.5 Comparison to SMT-CBMC

This subsection describes the evaluation of ESBMC against another SMT-based BMC tool developed by Armando et al. [3]. For the evaluation, we took the official benchmark of the SMT-CBMC tool [35]; Table 3 summarizes the results. The timings in brackets again refer to the SMT-LIB interface. Note that results given for ESBMC differ from those in Table 1: since SMT-CBMC does not generate any checks for safety properties we used both systems only to check the single user-specified property. SMT-CBMC has been invoked by setting manually the file name and the unwinding bound (i.e., SMT-CBMC `-file` *Module* `-bound` *B*). Furthermore, we compared SMT-CBMC with its default solver (i.e, CVC3 2.2) against ESBMC using both its default solver (i.e., Z3 2.11) as well as CVC3 2.2.

If CVC3 is used as the SMT solver, both tools run out of memory and thus fail to analyze *BubbleSort* for large N (N=140). SMT-CBMC runs out of time when analyzing the program *SelectionSort* and *StrCmp* while ESBMC runs out of time for the program *MinMax*. ESBMC outperforms SMT-CBMC by a factor of 6-90 for those benchmarks that do not fail. However, if Z3 is used as solver for ESBMC, the difference between both tools becomes more noticeable and ESBMC generally outperforms SMT-CBMC by a factor of 10-200.

4.6 Comparison to SAT-based CBMC

CBMC [11] is one of the most widely used BMC tools for ANSI-C. It has recently been extended by an SMT backend [6], and in our comparison we tried to use the SMT solvers Z3 and Boolector (by invoking `--z3` or `--boolector`) for evaluating both tools CBMC and ESBMC. However, the SMT-based CBMC version failed to check all benchmarks reported in Table 4 due to problems in the SMT back-end. Consequently, we compare

	Testsuite	N	ΣL	ΣP	Time		Properties		Errors		
					Solver	Total	Passed	Violated	N_e	true	false
1	EUREKA	8	821	437	224	755	437	0	-	-	-
2	NECLA-ok	30	891	254	98	172	212	3	2	3	0
	NECLA-bad	10	342	112	37	47	87	25	10	25	0
3	VERISEC-ok	80	4521	2114	128	211	2094	15	9	15	0
	VERISEC-bad	83	4569	2024	127	226	1808	216	83	216	0
4	PowerStone	9	2857	2031	728	816	2014	17	1	12	0
5	SNU-RT	20	3320	828	15	570	799	29	4	29	0
6	WCET	10	3430	726	7	73	722	4	2	3	1

TABLE 2: Results of the error-detection capability of ESBMC.

	Module	L	B	P	ESBMC (Z3)		ESBMC (CVC3)		SMT-CBMC [3]
					Solver	Total	Solver	Total	Total
1	EUREKA.BubbleSort	43	35	1	<1 (<1)	**2 (2)**	15 (3)	16 (3)	100
		43	70	1	3 (1)	**13** (15)	M_b (16)	M_b (30)	407
		43	140	1	68 (11)	**259** (265)	M_b (M_b)	M_b (M_b)	M
2	EUREKA.SelectionSort	34	35	1	<1 (<1)	**<1 (<1)**	<1 (<1)	**<1 (<1)**	T
		34	70	1	<1 (<1)	**8** (9)	<1 (7)	**8** (15)	T
		34	140	1	10 (4)	**157** (162)	2 (34)	160 (193)	T
3	EUREKA.BellmanFord	49	20	1	<1 (<1)	**<1 (<1)**	<1 (<1)	**<1 (<1)**	43
4	EUREKA.Prim	79	8	1	<1 (<1)	**<1 (<1)**	<1 (<1)	**<1 (<1)**	96
5	EUREKA.StrCmp	14	1000	1	25 (30)	**27** (38)	3 (253)	7 (261)	T
6	EUREKA.SumArray	12	1000	1	9 (<1)	25 **(<1)**	**<1** (108)	<1 (108)	98
7	EUREKA.MinMax	19	1000	1	2 (1)	**6 (6)**	T_b (M_b)	T_b (M_b)	65

TABLE 3: Results of the comparison between ESBMC and SMT-CBMC.

our approach only against the SAT-based CBMC version, which is able to support most of the benchmarks from Table 4; in particular, compared CBMC v3.8 and ESBMC v1.15. We invoked both tools by manually setting the file name, the unwinding bound, the checks for array bounds, pointer safety, division by zero, and arithmetic over- and underflow.[2] Table 4 reports the results in the usual format.

As we can see in Table 4, SAT-based CBMC is not able to check the module *pocsag* due to memory limitations; it times out in five cases and fails in four cases due to errors in the front-end, and in another five cases due to errors in the back-end. ESBMC runs out of time to check the modules *qurt* and *ludcmp*, but it is able to check seven (of eight) properties of the module *qurt* and fifteen additional benchmarks in comparison to SAT-based CBMC. Both CBMC and ESBMC find errors in the SNU-RT (as confirmed in Section 4.4) and NXP benchmark suites. However, ESBMC finds additional confirmed errors (see Section 4.4 again) in the WCET, SNU-RT, and PowerStone benchmarks, while CBMC produces false negatives or fails. In the case of *print_tokens2*, ESBMC runs out of memory if we try to increase the unwinding bound to 82, but if we restrict the verification to the function *get_token*, it finds an array-bounds violation in the golden version. We extracted the counterexample provided by ESBMC and used it to confirm that this is a true fault. ESBMC also finds additional errors in *flasher_manager* (violation of a user-specified assertion), *exStbHwAcc* (arithmetic overflow on typecast), and *adpcm_encode* (array-bounds violation) applications. Moreover, SAT-based CBMC also produces false negatives for the golden version of the programs *ex30* and *ex33* by reporting non-existing bugs related to dynamic object upper bounds and invalid pointers. We can also see that ESBMC not only has a better precision than SAT-based CBMC, but it also runs slightly faster than the SAT-based CBMC in those benchmarks that it does not fail. The results in Table 4 thus allow us to assess quantitatively that ESBMC improves substantially precision and scales significantly better than CBMC for problems that involve tight interplay between non-linear arithmetic, bit operations, pointers and array manipulations, which are typical for embedded systems software.

2. The tools where invoked as follows: `cbmc file --unwind` *B* `--bounds-check --div-by-zero-check --pointer-check --overflow-check --string-abstraction` and `esbmc file --unwind` *B* `--overflow-check --string-abstraction`.

5 Related Work

SMT-based BMC is gaining popularity in the formal verification community due to the advent of sophisti-

					SAT-based CBMC (v3.8) [11]					ESBMC (v1.15)				
					Time		Properties			Time		Properties		
	Module	L	B	P	Solver	Total	Passed	Violated	Fail	Solver	Total	Passed	Violated	Fail
1	Siemens.print_tokens2	510	81*	135	<1	<1	135	0	0	<1 (<1)	<1 (<1)	135	0	0
	(get_token)	51	82	76	T_b	T_b	0	0	135	29 (35)	**60** (65)	134	1	0
2	Siemens.replace	564	1*	199	$\dagger_f$	$\dagger_f$	-	-	-	<1 (<1)	**<1 (<1)**	199	0	0
3	Siemens.tot_info	406	30*	73	$\dagger_f$	$\dagger_f$	-	-	-	32 (3)	98 **(79)**	73	0	0
4	Siemens.tcas	173	4	38	<1	**<1**	38	0	0	1 (<1)	2 (1)	38	0	0
5	Siemens.space	9125	126*	2016	<1	4	2016	0	0	<1 (<1)	**3 (3)**	2016	0	0
6	WCET.statistics	157	∞	29	$\dagger_f$	$\dagger_f$	-	-	-	1 (<1)	**53 (53)**	27	2	0
7	WCET.statemate	1273	3	6	<1	<1	6	0	0	<1 (<1)	<1 (<1)	6	0	0
8	SNU-RT.crc_new	125	∞	13	<1	**6**	12	1	0	<1 (<1)	8 (8)	12	1	0
9	SNU-RT.fft1k_new	158	∞	39	$\dagger_b$	$\dagger_b$	35	0	4	<1 (1)	**56** (57)	39	0	0
10	SNU-RT.fibcall_new	83	50*	2	<1	<1	1	1	0	<1 (<1)	<1 (<1)	1	1	0
11	SNU-RT.fir_new	316	∞	25	5	6	25	0	0	<1 (<1)	**2 (2)**	25	0	0
12	SNU-RT.insertsort_new	94	13	20	$\dagger_b$	$\dagger_b$	0	0	20	8 (<1)	8 **(2)**	14	6	0
13	SNU-RT.lms_new	256	∞	35	$\dagger_b$	$\dagger_b$	29	0	6	3 (<1)	**24 (24)**	35	0	0
14	SNU-RT.ludcmp_new	142	∞	79	T_b	T_b	84	0	4	T_b (T_b)	T_b (T_b)	84	0	4
15	SNU-RT.qurt_new	159	∞	8	T_b	T_b	2	0	6	T_b (T_b)	T_b (T_b)	7	0	1
16	PowerStone.bcnt	83	17	153	2	3	153	0	0	2 (2)	**2 (2)**	153	0	0
17	PowerStone.blit	95	1	133	<1	<1	133	0	0	<1 (<1)	<1 (<1)	129	4	0
18	PowerStone.pocsag	521	42	187	M_f	M_f	-	-	-	4 (<30)	**22** (48)	186	1	0
19	NECLA.ex30	45	101	16	<1	**2**	12	4	0	<1 (<1)	3 (3)	16	0	0
20	NECLA.ex33	35	100	13	<1	<1	6	7	0	<1 (<1)	<1 (<1)	13	0	0
21	NXP.exStbKey	558	4	33	<1	4	33	0	0	<1 (<1)	**1 (1)**	33	0	0
22	NXP.exStbHDMI	1508	15*	138	500	706	138	0	0	316 ($\dagger_b$)	**429** ($\dagger_b$)	138	0	0
23	NXP.exStbLED	430	50*	102	72	122	102	0	0	48 (68)	80 **(79)**	102	0	0
24	NXP.exStbHwAcc	1432	3	239	2	6	238	1	0	<1 ($\dagger_b$)	**1** ($\dagger_b$)	238	1	0
25	NXP.exStbResolution	353	50	79	$\dagger_b$	$\dagger_b$	0	0	70	26 (59)	**59** (61)	70	0	0
26	NXP.exStbFb	689	10	218	484	825	167	0	0	52 ($\dagger_b$)	**101** ($\dagger_b$)	167	0	0
27	NXP.exStbCc	331	3	21	<1	3	19	2	0	<1 (<1)	**<1** (<1)	19	2	0
28	picosat	8160	23*	3142	T_f	T_f	-	-	-	27 ($\dagger_b$)	**79** ($\dagger_b$)	3142	0	0
29	flex	14192	2*	10002	$\dagger_f$	$\dagger_f$	-	-	-	3492 ($\dagger_b$)	**3526** ($\dagger_b$)	10002	0	0
30	git-remote-gitkrb5	6288	5*	174	$\dagger_b$	$\dagger_b$	0	0	174	196 ($\dagger_b$)	**225** ($\dagger_b$)	174	0	0
31	flasher_manager	521	21	26	2	**4**	26	0	0	25 (22)	29 (27)	25	1	0
32	HLS.adpcm_encode	150	100	25	T_b	T_b	0	0	25	<1 (<1)	**6 (6)**	24	1	0

TABLE 4: Results of the comparison between CBMC and ESBMC. Internal errors in the respective tool are represented with † in the Time column. The subscripts f and b indicate whether the errors occurred in the front-end or back-end, respectively. We give the smallest unwinding bound that is sufficient to prove or falsify the properties (i.e., produces no unwinding violation); a superscript $*$ on the unwinding bound indicates that the bound is insufficient, but cannot be increased with the available memory.

cated SMT solvers built over efficient SAT solvers [16], [17], [18]. Previous work related to SMT-based BMC [4], [48], [3] combined decision procedures for the theories of uninterpreted functions, arrays and linear arithmetic only, but did not encode key constructs of the ANSI-C programming language such as bit operations, floating-point arithmetic and pointers.

Ganai and Gupta describe a verification framework for BMC which extracts high-level design information from an extended finite state machine (EFSM) and applies several techniques to simplify the BMC problem [4], [24]. However, the authors flatten structures and arrays into scalar variables in such a way that they use only the theory of integer and real arithmetic in order to solve the VCs. Armando et al. also propose a BMC approach using SMT solvers for C programs [3]. However, they only make use of linear arithmetic (i.e., addition and multiplication by constants), arrays, records and bit-vectors in order to solve the VCs. As a consequence, their SMT-CBMC prototype does not address important constructs of the ANSI-C programming language such as non-linear arithmetic and bit-shift operations. Kroening

also encodes the VCs generated by the front-end of CBMC by using the bit-vector arithmetic and does not exploit other background theories of the SMT solvers to improve scalability [6]. Donaldson et al. present an approach to compute invariants in BMC of software by means of *k*-induction [49]. Their method, however, is highly customized for checking assertions representing DMA operations in the Cell processor, which requires only a small number of loop iterations and thus allows *k*-induction to work well with a small value of *k*. Xu proposes the use of SMT-based BMC to verify real-time systems by using TCTL to specify the properties [48]. The author considers an informal specification (written in English) of the real-time system and then models the variables using integers and reals and represents the clock constraints using linear arithmetic expressions.

De Moura et al. present a bounded model checker that combines propositional SAT solvers with domain-specific theorem provers over infinite domains [50]. Differently from other related work, the authors abstract the Boolean formula and then apply a lazy approach to refine it in an incremental way. This approach is applied to verify timed automata and RTL level descriptions. Jackson et al. [51] discharge several VCs from programs written in the Spark language to the SMT solvers CVC3 and Yices as well as to the theorem prover Simplify. The idea of this work is to replace the Praxis prover by CVC3, Yices and Simplify in order to generate counter-example witnesses to VCs that are not valid. In [52], Jackson and Passmore extend [51] by implementing a tool to automatically discharge VCs using SMT solvers. The authors observed significant performance improvement of the SMT solvers when compared to the Praxis prover. Jackson and Passmore, however, focus on translating VCs into SMT from programs written in the SPARK language (which is a subset of the Ada language) instead of ANSI-C programs.

Software model checking is executed on an abstraction of the actual program. Model checking the abstraction of a program is sound, but necessarily incomplete. Abstraction refinement is a general technique for proving properties with software model checkers [60]. Thus, abstraction refinement allows extending the usual bug-hunting uses of software model checkers. In practice, a well-known approach is counterexample-guided abstraction refinement (CEGAR) [64]. A number of approaches have been developed for CEGAR [60], including interpolation [63]. Examples of modern software model checkers implementing CEGAR with interpolation include BLAST [62] and ARMC [61].

Recently, a number of static checkers have been developed in order to trade off scalability and precision. Calysto is an automatic static checker that is able to verify VCs related to arithmetic overflow, null-pointer dereferences and assertions specified by the user [53]. The VCs are passed to the SMT solver SPEAR which supports boolean logic, bit-vector arithmetic and is highly customized for the VCs generated by Calysto. However, Calysto does not support floating-point operations and unsoundly approximates loops by unrolling them only once. As a consequence, soundness is relinquished for performance. Saturn is another automatic static checker that scales to larger systems, but with the drawback of losing precision by supporting only the most common integer operators and performing at most two unwindings of each loop [54]. In contrast to [53], [54], the extended static checker for Java (ESC/JAVA) is a semi-automatic verification tool, which requires the programmer to supply loop, function, and class invariants and thus limits its acceptance in pratice [55]. In addition, ECS/Java employs the Simplify theorem prover [56] to verify user-supplied invariants and thus important constructs of the programming language (e.g., bitwise operation) are often encoded imprecisely using axioms and uninterpreted functions.

6 Conclusions

In this work, we have investigated SMT-based verification of ANSI-C programs, with a focus on embedded software. We have described a new set of encodings that allow us to reason accurately about bit operations, unions, fixed-point arithmetic, pointers and pointer arithmetic and implemented it in the ESBMC tool. Our experiments constitute, to the best of our knowledge, the first substantial evaluation of SMT-based BMC on industrial applications. The results show that ESBMC outperforms CBMC [11] and SMT-CBMC [3] if we consider the verification of embedded software. ESBMC is able to model check ANSI-C programs that involve tight interplay between non-linear arithmetic, bit operations, pointers and array manipulations. In addition, it was able to find undiscovered bugs in the NECLA, PowerStone, Siemens, SNU-RT, VERISEC and WCET benchmarks related to arithmetic overflow, buffer overflow, invalid pointers and pointer arithmetic.

SMT-CBMC still has limitations not only in the verification time (due to the lack of simplification based on high-level information), but also in the encodings of important ANSI-C constructs used in embedded software. CBMC is a SAT-based BMC tool for full ANSI-C, but it has limitations due to the fact that the size of the propositional formulae increases significantly in the presence of large data-paths and high-level information is lost when the VCs are converted into propositional logic (preventing potential optimizations to reduce the state space to be explored). Its prototype SMT-based back-end is still unstable and fails on a large fraction of our benchmarks.

We are currently extending ESBMC to support the verification of multi-threaded software in embedded systems [57], [58]. For future work, we also intend to investigate the application of termination analysis [59] and incorporate reduction methods to simplify the *k*-model.

Acknowledgments

We thank D. Kroening, C. Wintersteiger, and L. Platania for many helpful discussions about the CBMC and SMT-CBMC model checking tools, C. Barrett, R. Brummayer and L. de Moura for analyzing the VCs, and F. Ivancic and M. Chechik for checking the bugs discovered in the NECLA and VERISEC suites. We also thank the anonymous reviewers for their comments, which helped us to improve the draft version of this paper.

This research was supported by EPSRC grants EP/E012973/1 (NOTOS) and EP/F052669/1 (Cadged Code) and by the EC FP7 grants ICT/217069 (COCONUT) and IST/033709 (VERTIGO). Lucas Cordeiro was also supported by an ORSAS studentship.

REFERENCES

[1] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pp. 457–481. 2009.
[2] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," in *SPIN*, LNCS 3925, pp. 146–162, 2006.
[3] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," in *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, 2009.
[4] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 794–801, 2006.
[5] ISO, *ISO/IEC 9899:1999: Programming languages C*, International Organization for Standardization, 1999.
[6] D. Kroening, *CBMC 3.3 released – preliminary support for SMT QF_AUFBV*. http://groups.google.co.uk/group/cprover: The CProver Group, 2009.
[7] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," in *Intl. Conf. on Automated Software Engineering (ASE)*, pp. 137–148, 2009.
[8] ——, "Continuous verification of large embedded software using SMT-based bounded model checking." in *Intl. Conf. and Workshops on Engineering of Computer-Based Systems (ECBS)*, pp. 160–169, 2010.
[9] SMT-LIB, *The Satisfiability Modulo Theories Library*, http://combination.cs.uiowa.edu/smtlib, 2009.
[10] A. Stump and M. Deters, *Satisfiability Modulo Theories Competition*, http://www.smtcomp.org/, 2010.
[11] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2988, pp. 168–176, 2004.
[12] C. Wintersteiger, *Compiling GOTO-Programs*, http://www.cprover.org/goto-cc/, 2009.
[13] R. L. Sites, "Some thoughts on proving clean termination of programs." Stanford, CA, USA, Tech. Rep., 1974.
[14] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.
[15] M. Bozzano and et al. "Encoding RTL constructs for MathSAT: a preliminary report," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 2, pp. 3–14, 2006.
[16] C. Barrett and C. Tinelli, "CVC3," in *Intl. Conf. on Computer Aided Verification (CAV*, LNCS 4590, pp. 298–302, 2007.
[17] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 5505, pp. 174–177, 2009.
[18] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 4963, pp. 337–340, 2008.
[19] J. Mccarthy, "Towards a mathematical science of computation," in *IFIP Congress*. North-Holland, pp. 21–28, 1962.
[20] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.
[21] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Computational challenges in bounded model checking," *Software Tools for Technology Transfer (STTT)*, vol. 7, no. 2, pp. 174–183, 2005.
[22] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Brazilian Symposium on Formal Methods (SBMF)*, LNCS 5902, pp. 23–36, 2009.
[23] E. Clarke, D. Kroening, O. Strichman, and J. Ouaknine, "Completeness and complexity of bounded model checking," in *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS 2937, pp. 85–96, 2004.
[24] M. K. Ganai and A. Gupta, "Completeness in SMT-based BMC for software programs," in *Design, Automation, and Test in Europe (DATE)*, IEEE, pp. 831–836, 2008.
[25] A. S. Tanenbaum, *Computer networks: 4th edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2002.
[26] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT," in *Formal Methods in System Design (FMSD)*, vol. 25, pp. 105–127, 2004.
[27] S.-S. Lim, *Seoul National University Real-Time Benchmarks Suite*, http://archi.snu.ac.kr/realtime/benchmark/, 2009.
[28] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
[29] D. Gries and G. Levin, "Assignment and procedure call proof rules," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 4, pp. 564–579, 1980.
[30] D. Kroening, E. Clarke, and K. Yorav, "Behavioral consistency of C and Verilog programs using bounded model checking," in *Technical Report, CMU-CS-03-126*, 2003.
[31] J. A. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Intl. Conf. on Software Engineering (ICSE) (1)*, pp. 515–524, 2010.
[32] D. Kroening and S. A. Seshia, "Formal verification at higher levels of abstraction," in *Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 572–578, 2007.
[33] S. Gupta, *High Level Synthesis Benchmarks Suite*, http://mesl.ucsd.edu/spark/benchmarks.shtml, 2009.
[34] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A buffer overflow benchmark for software model checkers," in *Intl. Conf. on Automated Software Engineering (ASE)*, pp. 389–392, 2007.
[35] L. Platania, *Eureka Benchmark Suite*, http://www.ai-lab.it/eureka/bmc.html, 2009.
[36] S. Sankaranarayanan, *NECLA Static Analysis Benchmarks*. http://www.nec-labs.com/research/system/, 2009.
[37] J. Scott, L. H. Lee, A. Chin, J. Arends, and B. Moyer, "Designing the low-power m*core architecture," in *Intl. Symp. Computer Architecture Power Driven Microarchitecture Workshop*, IEEE, pp. 145–150, 1998.
[38] A. Ermedahl and J. Gustafsson, *Worst-case execution time project / Benchmarks*, http://www.mrtc.mdh.se/projects/wcet/, 2009.
[39] T. Ostrand, *Siemens Corporate Research*, http://sir.unl.edu/portal/, 2010.
[40] NXP, *High definition IP and hybrid DTV set-top box STB225*. http://www.nxp.com/, 2009.
[41] *MiBench Version 1.0*, http://www.eecs.umich.edu/mibench/, 2009.
[42] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, no. 2-4, pp. 75–97, 2008.
[43] M. Ramanathan, *flex*, http://sir.unl.edu/portal/, 2010.
[44] J. Morse, *Kerberos Git*, https://www.studentrobotics.org/trac/wiki/Kerberos/Git, 2011.
[45] N. L. Vinh, *The Flasher Manager Application*, http://users.polytech.unice.fr/ rueher/Benchs/FM/, 2010.
[46] F. Ivancic, *Personal communication*, 2011.
[47] M. Chechik, *Personal communication*, 2011.
[48] L. Xu, "SMT-based bounded model checking for real-time systems," in *Intl. Conf. on Quality Software (QSIC)*, IEEE, pp. 120–125, 2008.
[49] A. Donaldson, D. Kroening, and P. Rümmer, "Automatic analysis of scratch-pad memory code for heterogeneous multicore processors," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 6015, pp. 280–295, 2010.
[50] L. M. de Moura, H. Rueß, and M. Sorea, "Lazy theorem proving for bounded model checking over infinite domains," in *Intl. Conf. on Automated Deduction (CADE)*, LNCS 2392, pp. 438–455, 2002.
[51] P. B. Jackson, B. J. Ellis, and K. Sharp, "Using SMT solvers to verify high-integrity programs," in *2nd Workshop on Automated Formal Methods*, pp. 60–68, 2007.
[52] P. B. Jackson and G. O. Passmore, *Proving SPARK Verification Conditions with SMT solvers*. Technical Report, University

of Edinburgh, http://homepages.inf.ed.ac.uk/pbj/papers/vct-dec09-draft.pdf, 2009.
[53] D. Babić and A. J. Hu, "Calysto: Scalable and Precise Extended Static Checking," in *Intl. Conf. on Software Engineering (ICSE)*, pp. 211–220, 2008.
[54] Y. Xie and A. Aiken, "Scalable error detection using Boolean satisfiability," *Special Interest Group on Programming Languages (SIGPLAN) Not.*, pp. 351–363, 2005.
[55] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Programming Language Design and Implementation (PLDI)*, pp. 234–245, 2002.
[56] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.
[57] L. Cordeiro, "SMT-based bounded model checking for multi-threaded software in embedded systems." in *Intl. Conf. on Software Engineering (ICSE), Doctoral Symposium*, pp. 373–376, 2010.
[58] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," *To appear in 33rd Intl. Conf. on Software Engineering (ICSE)*, 2011.
[59] R. C. Andreas, B. Cook, A. Podelski, and A. Rybalchenko, "Terminator: Beyond safety," in *Intl. Conf. on Computer Aided Verification (CAV)*, LNCS 4144, pp. 415–418, 2006.
[60] R. Jhala, R. Majumdar, "Software model checking," in *ACM Comput. Surv.*, vol. 41, no. 4, pp. 1–54, 2009.
[61] A. Podelski, A. Rybalchenko, "ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement," in *Practical Aspects of Declarative Languages (PADL)*, pp. 245–259, 2007.
[62] D. Beyer, T. Henzinger, R. Jhala, R. Majumdar, "The software model checker Blast," in *Int. J. Softw. Tools Technol. Transf.* vol. 9, no. 5-6, pp. 505-525, 2007.
[63] K. McMillan. Interpolation and sat-based model checking. in *Intl. Conf. on Computer Aided Verification (CAV)*, *LNCS* 2725, pages 1–13, 2003.
[64] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, "Counterexample-Guided Abstraction Refinement," in *Intl. Conf. on Computer Aided Verification (CAV)*, *LNCS* 1855, pp. 154-169

Lucas Cordeiro Lucas Cordeiro received the B.Sc. degree in electrical engineering and the M.Sc. degree in computer engineering from the Federal University of Amazonas (UFAM), in 2005 and 2007, respectively. He received the Ph.D. degree in computer science from University of Southampton in 2011. Currently, he is an assistant professor in the Electronic and Information Research Center at UFAM. His work focuses on software verification, bounded (and unbounded) model checking, satisfiability modulo theories and embedded systems.

Bernd Fischer Bernd Fischer received his PhD degree in Computer Science in 2001 from the University of Passau, Germany. From 1998 to 2006, he was a research scientist with USRA/RIACS at the NASA Ames Research Center. Since 2006 he is a Senior Lecturer for computer science at the University of Southampton. His current research interests include code generation, programming languages, formal methods, software reliability, and software verification.

Joao Marques-Silva Joao Marques-Silva (M'95-SM'03) received a Ph.D. degree from the University of Michigan, Ann Arbor, USA, in 1995, and the Habilitation degree in computer science from the Technical University of Lisbon in 2004. He is currently Stokes Professor of Computer Science and Informatics, University College Dublin (UCD), Ireland. He is also Professor of Computer Science at Instituto Superior Tecnico (IST), Portugal. His research interests include algorithms for constraint solving and optimization, and applications in formal methods, artificial intelligence, operations research, and bioinformatics.