Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers

Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania

Artificial Intelligence Laboratory
DIST, Università degli Studi di Genova
Viale F. Causa 13, 16145 Genova, Italy
{armando, jacopo, lorenzo}@dist.unige.it

Abstract. C Bounded Model Checking (CBMC) has proven to be a successful approach to automatic software analysis. The key idea is to *(i)* build a propositional formula whose models correspond to program traces (of bounded length) that violate some given property and *(ii)* use state-of-the-art SAT solvers to check the resulting formulae for satisfiability. In this paper we propose a generalisation of the CBMC approach based on an encoding into richer (but still decidable) theories than propositional logic. We show that our approach may lead to considerably more compact formulae than those obtained with CBMC. We have built a prototype implementation of our technique that uses a Satisfiability Modulo Theories (SMT) solver to solve the resulting formulae. Computer experiments indicate that our approach compares favourably with and on some significant problems outperforms CBMC.

1 Introduction

SAT-based Bounded Model Checking (BMC) [1] was originally proposed as a complementary technique to OBDD-based model checking for the automatic analysis of finite state systems (e.g. hardware circuits). The key idea is to build a propositional formula whose models correspond to behaviours of the system that violate a given property.

The application of Bounded Model Checking to software poses new challenges, as most programs are inherently infinite-state and new, non trivial issues such as the handling of (recursive) function calls and the modelling of complex data structures must be properly addressed. An elegant solution to the problem is proposed in [2, 3] and implemented in the CBMC (C Bounded Model Checking) model checker. The approach amounts to *(i)* building a propositional formula whose models correspond to program traces (of bounded length) violating some given property and *(ii)* using state-of-the-art SAT solvers to check the resulting formulae for satisfiability.

In this paper we propose a generalisation of the CBMC approach. Instead of encoding the program into a propositional formula, we encode it into a quantifier-free formula to be checked for satisfiability w.r.t. some given decidable theory

(henceforth called *background theory*) and use a state-of-the-art SMT (Satisfiability Modulo Theories) solver to perform the satisfiability checking.

We show that our approach may lead to considerably more compact formulae when arrays are involved in the input program. In particular the size of the formulae generated by our approach does not depend on the size of the bit-vector representation of the basic data types nor on the size of the arrays occurring in the program, whereas the encoding technique implemented in CBMC depends on both.

Experimental results obtained with a prototype implementation of our technique, called SMT-CBMC, confirm the effectiveness of our approach: on a number of problems involving complex interactions of arithmetics and arrays manipulation CBMC generates formulae whose size makes the solving phase impractical. On the other hand, SMT-CBMC scales significantly better than CBMC as the size of the arrays occurring in the input program increases.

Structure of the paper. In the next section (Section 2) we provide a brief introduction to SMT and present a set of decidable theories that we will refer to in the rest of the paper. In Section 3 we present our generalisation to the CBMC approach: we describe the generation of the formula, the different approaches to solve the formula, and how error traces are reconstructed by exploiting the information returned by the SMT solver. In Section 4 we describe our prototype tool SMT-CBMC and present the experimental results. In Section 5 we discuss the related work and finally, in Section 6, we draw some concluding remarks.

2 Satisfiability Modulo Theories

Given a decidable theory $\mathcal{T}$ and a quantifier-free formula ϕ in the same language as $\mathcal{T}$, we say that ϕ *is* $\mathcal{T}$*-satisfiable* if and only if there exists a model of $\mathcal{T}$ which is also a model of ϕ or, equivalently, if $\mathcal{T} \cup \{\phi\}$ is satisfiable. A *SMT solver for* $\mathcal{T}$ is a program capable of determining the $\mathcal{T}$-satisfiability of every quantifier-free formula ϕ in the same language as $\mathcal{T}$. Let $\Gamma \cup \{\phi\}$ be a set of formulae in the same language as $\mathcal{T}$, we say that ϕ *is a* $\mathcal{T}$*-consequence of* Γ, in symbols $\Gamma \models_{\mathcal{T}} \phi$, if and only if every model of $\mathcal{T} \cup \Gamma$ is a model of ϕ. Obviously, the problem of determining whether $\mathcal{C} \models_{\mathcal{T}} \phi$ holds can be reduced to the problem of checking the $\mathcal{T}$-satisfiability of $\mathcal{C} \cup \{\neg\phi\}$.

Over the last three decades, a great deal of attention has been paid to solving the SMT problem for a number of (decidable) theories of interest such as, e.g., Linear Arithmetics, the theory of lists, the theory of arrays, and—more recently—the theory of bit-vectors. The practical relevance of these theories in verification cannot be overestimated as arithmetics, lists, arrays, and bit-vectors are ubiquitous in Computer Science. Moreover, since these entities rarely occur in isolation, the problem of building SMT solvers for the combination of two (or more) decidable theories (say $\mathcal{T}_1 \cup \mathcal{T}_2$) out of SMT solvers for the component theories (say $\mathcal{T}_1$ and $\mathcal{T}_2$) has also been thoroughly investigated and solutions identified

[4, 5]. More recently the problem of combining the effectiveness of state-of-the-art SAT solvers with SMT solvers has received growing attention and has led to a new generation of SMT solvers capable of remarkable performance [6].

In the rest of this section we give a brief description of the decidable theories that are relevant for the present paper.

Linear Arithmetics. By Linear Arithmetics we mean standard arithmetics (either over $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$) with addition (i.e. $+$) and the usual relational operators (e.g. $=$, $<$, $\leq$, $>$, $\geq$) but without multiplication. Multiplication by a constant, say $n * x$ where n is a numeral, is usually allowed but it is just a notational shorthand for the (linear) expression $x + \cdots + x$ with n occurrences of the variable x.

The theory of arrays. Arrays are data structures representing arbitrary associations of elements to a set of indexes. Unlike arrays available in standard programming languages, the arrays modelled by the theory of arrays need not have finite size. Given sorts INDEX, ELEM and ARRAY for indices, elements, and arrays (resp.) and function symbols select : ARRAY $\times$ INDEX $\rightarrow$ ELEM and store : ARRAY $\times$ INDEX $\times$ ELEM $\rightarrow$ ARRAY, the standard presentation of the theory of arrays consists of the following two axioms:

$$\begin{aligned} &\forall a, i, e. && \text{select}(\text{store}(a, i, e), i) = e \\ &\forall a, i, j, e. && (i \neq j \supset \text{select}(\text{store}(a, i, e), j) = \text{select}(a, j)) \end{aligned}$$

with variable a of sort ARRAY, i and j of sort INDEX, and e of sort ELEM.

SMT solvers for the theory of arrays are described in [7, 8].

The theory of records. Records are data structures that aggregate attribute-value pairs. Let $Id = \{id_1, \ldots, id_n\}$ be a set of field identifiers and $\mathrm{T}_1, \ldots, \mathrm{T}_n$ be types, $\text{REC}(id_1 : \mathrm{T}_1, \ldots, id_n : \mathrm{T}_n)$, henceforth abbreviated REC, is the sort of records that associate an element of type T_k to the field identifier id_k, for $k = 1, \ldots, n$. The signature of the theory of records consists of a pair of function symbols $\text{rselect}_k : \text{REC} \rightarrow \mathrm{T}_k$ and $\text{rstore}_k : \text{REC} \times \mathrm{T}_k \rightarrow \text{REC}$ for $k = 1, \ldots, n$. The theory is finitely presented by the following axioms:

$$\begin{aligned} &\forall r, e. && \text{rselect}_k(\text{rstore}_k(r, e)) = e && \text{for } k = 1, \ldots, n \\ &\forall r, e. && \text{rselect}_l(\text{rstore}_k(r, e)) = \text{rselect}_l(r) && \text{for } k, l \text{ such that } 1 \leq k \neq l \leq n \end{aligned}$$

where r has sort REC and e has sort T_i.

A SMT solver for the theory of records is described in [9].

The theory of bit-vectors. Similarly to arrays, bit-vectors associate elements to a set of indexes, but unlike arrays the set of indexes is finite. Moreover the element associated to each index is boolean valued. Many theories of bit-vectors have been proposed in the literature [10–14], the main difference being whether bit-vectors are allowed to have variable size or not. For our purposes, the theory of fixed-size bit-vectors does suffice. The theory we consider has a sort $\text{BV}(n)$ for each positive integer n and a rich family of functions symbols consisting of

- *word-level functions*, e.g. $\_[i{:}j] : \text{BV}(m) \to \text{BV}(j-i+1)$ (bit-vector extraction) for $0 \leq i \leq j \leq m$, $@ : \text{BV}(m) \times \text{BV}(n) \to \text{BV}(m+n)$ (bit-vector concatenation) for $m, n > 0$;
- *bitwise functions*, e.g. $\tilde{}: \text{BV}(n) \to \text{BV}(n)$ (bitwise not), $\& : \text{BV}(n) \times \text{BV}(n) \to \text{BV}(n)$ (bitwise and), $| : \text{BV}(n) \times \text{BV}(n) \to \text{BV}(n)$ (bitwise or) for $n > 0$;
- *arithmetic functions*, e.g. $+ : \text{BV}(n) \times \text{BV}(n) \to \text{BV}(n)$ (addition modulo 2^n) for $n > 0$.

3 Bounded Model Checking of Sequential Software

As in [2], preliminarily to the generation of the formula, we preprocess the input program (Section 3.1). Given a bound $n > 0$, this amounts to applying a number of transformations which lead to a simplified program whose execution traces have finite length and correspond to the (possibly truncated) traces of the original program. The quantifier-free formula is then obtained by generating a quantifier-free formula for each statement of the resulting program (Section 3.2) and the resulting formula is fed to a SMT solver (Section 3.3). If an execution path leading to a violation of an `assert` statement occurring in the original program is detected, then a corresponding trace is built and returned to the user for inspection (Section 3.4).

In order to simplify the presentation, we assume that `=` is the only assignment operator occurring in the program and that no pointer variables nor conditional expressions occur in the program. Notice that all these simplifying assumptions can be readily lifted as in [15].

3.1 The Preprocessing Phase

The preprocessing activity starts by replacing `break` and `continue` statements with semantically equivalent `goto` statements. The `switch` construct is replaced by a proper combination of `if` and `goto` statements. Loops are then unwound by reducing them to a sequence of nested `if` statements. For instance, `while` loops are removed by applying the following transformation n times:

$$\texttt{while(}e\texttt{)\{ }I\texttt{ \}} \longrightarrow \texttt{if(}e\texttt{)\{ }I\texttt{ while(}e\texttt{)\{ }I\texttt{ \}\}}$$

The last `while` loop is finally replaced by the statement `assert(!`e`);`, called *unwinding assertion*. The failure of an unwinding assertion indicates that the bound n is not sufficient to model the system and the properties entirely, and that n must be increased.

Non recursive functions are then inlined. Recursive function calls and backward `goto`'s are unwound similarly to loop statements. Forward `goto` statements are transformed into equivalent `if` statements as explained in [1].

Next we put the program in Single Assignment Form [16] by renaming its variables in the following way. Let v be a program variable and i a program location. We define $\alpha(v, i)$ to be the number of assignments made to v prior to location i. Let e be a program expression. With $\varrho(e)$ we denote the expression obtained from

e by substituting every variable v in e with $v_{\alpha(v,i)}$. Every assignment to a variable x at a given location i, say x=e, is replaced by $x_{\alpha(x,i)+1}$=$\varrho(e)$. Every assignment to an array element, say a`[`e_1`]=`e_2, is replaced by $a_{\alpha(a,i)+1}$`[`$\varrho(e_1)$`]=`$\varrho(e_2)$. Every condition c (also called *guard*) of an `if` statement is replaced by $\varrho(c)$.

Then we put the program in *conditional normal form* by invoking the normalisation algorithm of Fig. 1 with G = `true` and by applying to G the usual simplifications. This normalisation step removes the `else` constructs and pushes

procedure NORMALISE(P,G)
  if P is an assignment or an `assert` statement **then**
    return `if`(G) P`;`
  else if P = (`if`(c) P_1`;`) **then**
    return NORMALISE(P_1,G `&&` c)
  else if P = (`if`(c) P_1`;else` $P2$`;`) **then**
    return NORMALISE(P_1,G `&&` c)`;`NORMALISE(P_2,G `&&(!` c))
  else if P = (P_1`;` $P2$) **then**
    return NORMALISE(P_1,G)`;`NORMALISE(P_2,G)
  end if
end procedure

Fig. 1. Turning the program in conditional normal form

the `if` statements downwards in the abstract syntax tree of the program till they are applied to atomic statements only. An example of the transformation of a program in conditional normal form is given in Fig. 2.

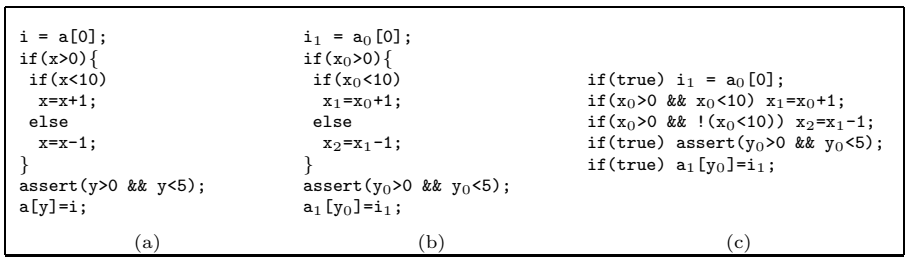


Fig. 2. Turning a program in conditional normal form: (a) the original program, (b) the renamed program, and (c) the normalised program.

Notice that a program in conditional normal form is a sequence of statements of the form `if`(c) s`;` where s is either an assignment or an `assert` statement.

Let P' be the program that is input to the normalisation algorithm, let n be the number of `assert` and assignment statements in P', and let m be the maximum number of atomic formulae occurring in the `if` guards. In the worst

case, the program output by the normalisation algorithm contains n `assert` and assignment statements guarded by n `if` guards each of which has at most $m \cdot log_2(n-1)$ atomic formulae. In fact, in the worst case P' is made of n `assert` and assignment statements and $n-1$ nested `if`...`else`... statements, that is, there are $n-1$ guards in P', each of which contains m atomic formulae.

3.2 The Encoding Phase

The application of the previous transformations leaves us with a renamed program P in conditional normal form. We now show how to build two sets of quantifier-free formulae $\mathcal{C}$ and $\mathcal{P}$ such that $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$ for some given background theory $\mathcal{T}$ if and only if no computation path of P violates any `assert` statement in P.

For each statement in P of the form `if`(c) v_{j+1}`=`e`;` $\mathcal{C}$ contains a formula of the form:[1]

$$v_{j+1} = (c' \ ? \ e' \ : \ v_j)$$

where c' and e' are obtained from c and e respectively by replacing every expression of the form a_l`[`e`]` with select(a_l, e). Similarly, for each statement in P of the form `if`(c) a_{j+1}`[`e_1`]=`e_2`;` $\mathcal{C}$ contains a formula of the form:

$$a_{j+1} \ = (c' \ ? \ \text{store}(a_j, e'_1, e'_2) \ : \ a_j) \tag{1}$$

where c', e'_1, and e'_2 are obtained from c, e_1, and e_2 respectively by substituting every every expression of the form a_l`[`e`]` with select(a_l, e).

The set $\mathcal{P}$ contains a formula of the form $(c' \supset e')$ for each statement in P at location i of the form `if`(c) `assert`(e)`;`, where c' and e' are obtained from c and e respectively by replacing every expression of the form a_l`[`e`]` with select(a_l, e).

$$\begin{aligned}
\mathcal{C} = \{ \ & i_1 = (\text{TRUE} \ ? \ \text{select}(a_0, 0) \ : \ i_0), \\
& x_1 = ((x_0 > 0 \wedge x_0 < 10) \ ? \ x_0 + 1 \ : \ x_0), \\
& x_2 = ((x_0 > 0 \wedge \neg(x_0 < 10)) \ ? \ x_1 - 1 \ : x_1), \\
& a_1 = (\text{TRUE}? \ \text{store}(a_0, y_0, i_1) \ : \ a_0)\} \\
\mathcal{P} \ = \ & \{\text{TRUE} \supset (y_0 > 0 \wedge y_0 < 5)\}
\end{aligned}$$

Fig. 3. The sets of formulae $\mathcal{C}$ and $\mathcal{P}$ for the program in Fig. 2.

[1] We use the expression $v = (c \ ? \ e_1 : e_2)$ as an abbreviation for the formula $(c \supset v = e_1) \wedge (\neg c \supset v = e_2)$.

3.3 Solving the Formula

Solving the Formula with a SAT Solver. In [2] this problem is reduced to a propositional satisfiability problem which is then fed to the Chaff SAT solver [17]. This is done by modelling variables of basic data types (e.g. `int` and `float`) as fixed-size bit-vectors and by considering the equations in $\mathcal{C}$ and in $\mathcal{P}$ as bit-vectors equations. Each array variable a is also replaced by $dim(a)$ distinct variables $a^0, \ldots, a^{dim(a)-1}$ and each formula of the form (1) occurring in $\mathcal{C}$ is replaced by the formula

$$\bigwedge_{i=0}^{dim(a)-1} a^i_{j+1} = ((c \wedge e_1 = i) \ ? \ e_2 : a^i_j).$$

Finally each term of the form $\text{select}(a_j, e)$ is replaced by a new variable, say x, and the following formulae are added to $\mathcal{C}$

$$\bigwedge_{i=0}^{dim(a)-1} ((e = i) \supset x = a^i_j)$$

The resulting set of bit-vector equations are then turned into a propositional formula. Variables of `struct` types are treated in a similar way. Notice that the size of the propositional formula generated in this way depends *(i)* on the size of the bit-vector representation of the basic data types as well as *(ii)* on the size of the arrays used in the program. It is worth pointing out that if the program contains a multi-dimensional array a with dimensions $d_1, \ldots, d_m$, then the number of added formulae grows as $O(d_1 \cdot d_2 \cdot \ldots \cdot d_m)$.

Solving the Formula with a SMT Solver. The alternative approach proposed in this paper is to use a SMT solver to directly check whether $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$. By proceeding in this way the size of the formula given as input to the SMT solver does not depend on the size of the bit-vector representation of the basic data types nor on the size of the arrays occurring in the program.[2] Moreover the use of a SMT solver gives us additional freedom in the way we model the basic data types. In fact, program variables with numeric type (e.g. `int`, `float`) can be modelled by variables ranging over bit-vectors or over the corresponding numerical domain (e.g. $\mathbb{Z}$, $\mathbb{R}$, resp.). If the modelling of numeric variables is done through fixed-size bit-vectors, then the result of the analysis is precise but it depends on the specific size considered for the bit-vectors. If, instead, the modelling of numeric variables is done through the corresponding numerical domain, then the result of the analysis is independent from the actual binary representation,

[2] It must be said that SMT solvers for the theory of bit-vectors may expand parts of the the formula by a technique known as bit-blasting, however this is usually done as a last resort and in many cases higher level and less expensive techniques are enough to solve the problem at hand [14].

but this comes to the price of losing completeness of the analysis if non linear expressions occur in the program.

In order to check check whether $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$, we use CVC Lite [18], a decision procedure that determines the validity of quantifier-free first-order formulae over the union of several theories.

3.4 Building the Error Trace

Whenever CVC Lite is asked to determine whether $\Gamma \models_{\mathcal{T}} \phi$, but this does not hold, the tool returns a finite set of formulae $\mathcal{K}$ such that $\Gamma, \mathcal{K} \models_{\mathcal{T}} \neg\phi$. The set of formulae $\mathcal{K}$ is said to be *a counterexample for* $\Gamma \models_{\mathcal{T}} \phi$.

Let $\mathcal{K}$ be a counterexample for $\mathcal{C} \models_{\mathcal{T}} \bigwedge \mathcal{P}$. We have defined a procedure that builds an error trace witnessing the violation of an `assert` statement occurring in the program P. The procedure (shown in Figure 4) traverses the control flow graph G_P of P starting from the first statement of P. Whenever a conditional statement `if`(e) is met, then the "then" branch is taken if $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} e$. If instead $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} \neg e$, then the "else" branch is taken. The *control flow graph* G_P *of* P is a directed graph $G_P = (N_P, Succ_P)$, where $N_P = \{1, \ldots, n\}$ is the set of vertices and $Succ_P : N_P \rightarrow 2^{N_P}$ maps each vertex in the set of its successors. For every vertex i such that $1 \leq \ i \leq n$, s_i denotes the program statement corresponding to i. By convention, node 1 of G_P denotes the first statement of P to be executed. If s_i is a conditional statement (i.e. it is of the form `if`(e)), then $Succ_P(i) = \{Tsucc_P(i), Fsucc_P(i)\}$, where $Tsucc_P(i)$ ($Fsucc_P(i)$) denotes the successor of i when e evaluates to true (false, resp.). If s_i is an assignment or an assertion statement, $Succ_P(i) = \{j\}$, with $j \in N_P$, and we define $sSucc_P(i) = j$.

Notice that lines 11–13 of the algorithm allow for the non deterministic selection of a branch of a conditional statement if neither $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} e$ nor $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} \neg e$ hold. This is necessary because the counterexample $\mathcal{K}$ might not be sufficient to determine the branch to choose. In this event, the branch can be chosen non deterministically, and any counterexample output by the algorithm is a valid one. Notice that this is a form of "don't care" non-determinism and therefore no backtracking is necessary. As an example of this, it can be noted that in the program of Figure 2 the assertion is violated independently from the value of variables `x` and `i`, and therefore also independently from the choice of the branches of the `if` statements. In fact, CVC Lite outputs a counterexample $\mathcal{K} = \{y_0 \geq 5\}$ for which neither $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} (x_0 > 0 \wedge x_0 < 10)$ nor $\mathcal{C}, \mathcal{K} \models_{\mathcal{T}} (x_0 > 0 \wedge \neg(x_0 < 10))$ hold.

4 Experimental Results

In order to assess the effectiveness of our approach we have developed a prototype implementation called SMT-CBMC. SMT-CBMC consists of four main modules, implemented in about 5,000 lines of Prolog code. The first module parses the input program, the second carries out the preprocessing, the third builds the quantifier-free formula, and the fourth module solves the formula according to

```
1: procedure ErrorTrace(i, K)
2:     if s_i is an assignment then
3:         Print("Assignment:", s_i)
4:         ErrorTrace(sSucc_P(i), K)
5:     else if s_i is if(e) then
6:         if C, K ⊨_T e then
7:             ErrorTrace(TSucc_P(i), K ∪ {e})
8:         else if C, K ⊨_T ¬e then
9:             ErrorTrace(FSucc_P(i), K ∪ {¬e})
10:        else
11:            Let ⟨j, c⟩ ∈ {⟨Tsucc_P(i), e⟩, ⟨Fsucc_P(i), ¬e⟩}
12:            be non-deterministically chosen.
13:            ErrorTrace(j, K ∪ {c})
14:        end if
15:    else if s_i is assert(e) then
16:        if C, K ⊨_T e then
17:            ErrorTrace(sSucc_P(i), K)
18:        else
19:            Print("Assertion violated:", assert(e))
20:            Halt
21:        end if
22:    end if
23: end procedure
```

Fig. 4. Building the program trace.

the user options by invoking CVC Lite.[3] The latter module also builds and prints the error trace whenever a counterexample is returned by CVC Lite.

We have run SMT-CBMC against a number of families of C programs. Each family of programs is parametric in a positive integer N and such that both the size of the arrays occurring in the programs and the number of iterations done by the programs depend on N. Therefore the instances become harder as the value of N increases. The benchmark problems considered are:

- `BubbleSort.c`(N), an implementation of the Bubble Sort algorithm [19],
- `SelectSort.c`(N), an implementation of the Selection Sort algorithm [19],
- `BellmanFord.c`(N), an implementation of the Bellman Ford algorithm [20, 21] for computing single-source shortest paths in a weighted graph, and
- `Prim.c`(N) an implementation of Prim's algorithm [22] for finding a minimum spanning tree for a connected weighted graph.

Notice that these programs are well-known and therefore the result of the analysis is not interesting in itself. However they allow us to carry out a systematic and quantitative assessment of the tools as the size of the arrays involved in the programs increases. It is also worth pointing out that all the benchmark problems considered involve a tight interplay between arithmetics and array manipulation.

[3] Currently SMT-CBMC can represent numeric data types with corresponding numeric domains as well as with fixed-size bit-vectors.

We have run both SMT-CBMC and CBMC on our benchmark programs. We report the total time spent by the tools to tackle each individual instance considered. Times are in seconds. All experiments have been obtained on a Pentium IV 2.4 GHz machine running Linux with the memory limit set to 800MB and the time limit set to 30 minutes. CBMC has been invoked by manually setting the unwinding bound (CBMC `--unwind n` option) and by disabling simplification (CBMC `--no-simplify` option).[4]

All the experiments presented in the rest of this section have been obtained by modelling the basic data types using bit-vectors thereby enabling the decision procedure for the theory of bit-vectors available in CVC Lite during the solving phase. Experimental results indicate that similar performances are obtained by modelling the numerical variables with the integers thereby enabling the decision procedure for linear arithmetics available in CVC Lite during the solving phase. In this section we report about testing the tools on safe instances of the benchmarks. Similar results are obtained if unsafe instances are considered.

More information about the experiments is available at URL `http://www.ai.dist.unige.it/eureka`.

4.1 Sorting algorithms

The Bubble Sort algorithm (see Figure 5) sorts the array `a` by using two nested loops that repeatedly swap adjacent elements. The assertion statements at the end of the program check that the array has been sorted. The parameter N here determines the size of the array, as well as the number of unwindings for each loop. Notice that in this case the number of unwindings grows quadratically with N as there are two nested loops.

The experimental results obtained for this family of programs are given in Figure 6. Plot (a) shows the time spent by the tools in analysing the program while plot (b) shows the size (in bytes) of the encodings. In both cases the value of N is on the the x-axis. CBMC runs out of memory for $N = 17$, while SMT-CBMC can still analyse programs for $N = 35$. A comparison between the formulae sizes of SMT-CBMC and CBMC substantiates our remarks about the size of the encodings: the formula built by SMT-CBMC for $N = 16$ is roughly two orders of magnitude smaller than the one built by CBMC.

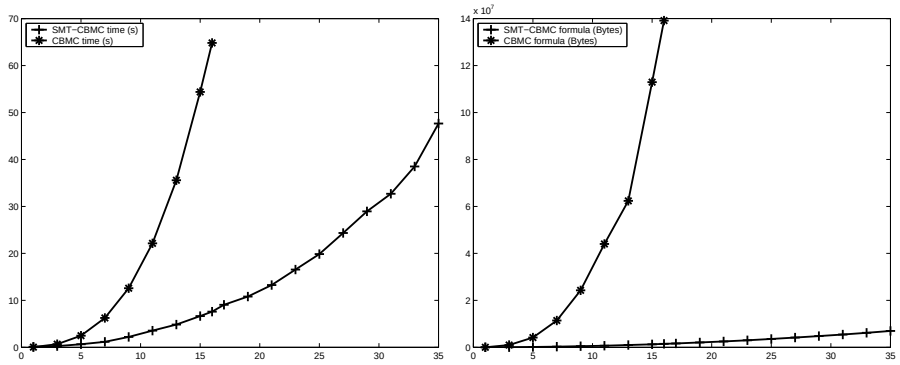
Similarly to Bubble Sort, Selection Sort (see Figure 7) counts two nested loops and a swap operation, and the assertions at the end of the program check that the given array has been sorted. Unlike Bubble Sort, where the swap is guarded by an `if` within the nested loop, the swap operation is done N times, where N is the size of the array, without any guard. Therefore, the encoding grows as $O(N \cdot dim(a))$, where $dim(a)$ is the size of the array. As shown in

[4] It is worth pointing out that CBMC features also an (undocumented) option `--cvc` whose effect is to output the bit-vector equations of the formula in the CVC format [23]. In this way it is possible to reason at the word-level, but still not using the theory of arrays. However this option is still experimental and not yet fully operational and therefore we have been unable to carry out experiments with it.

```
int a[N];
void main(){
 int i;
 a={N-1,...,0};
 BubbleSort();
 for(i=0;i<N;i++)
  assert(a[i]==i);
}
```

```
void BubbleSort(){
 int i,j,t;
 for (j=0;j<N-1;j++){
  for (i = 0; i< N-j-1; i++){
  if (a[i]>a[i+1]){
   t = a[i];
   a[i] = a[i+1];
   a[i+1] = t;
  }
 }
}
```

Fig. 5. Source code of `BubbleSort.c`(N)



(a) Time spent by the tools (b) Size of the formulae

Fig. 6. Results on `BubbleSort.c`(N)

plot (b) of Figure 8, CBMC runs out of memory already for $N = 21$, whereas SMT-CBMC analyses instances till $N = 35$.

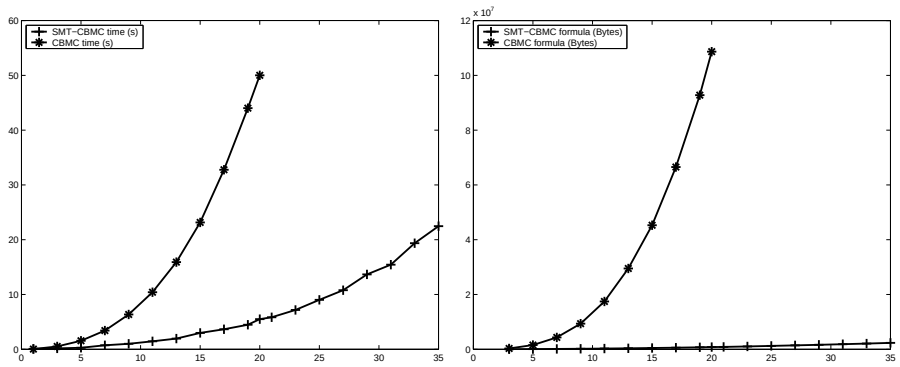
4.2 The Bellman-Ford algorithm

The problems of the `BellmanFord`(N) family model are implementations of the Bellman Ford algorithm with a graph comprising 5 nodes and N (randomly generated) edges. Each edge is associated with a (randomly generated) positive weight. The instance for $N = 5$ is given in Fig. 9. The edges are represented by the arrays `Source` and `Dest`, the weights by the array `Weight`. The `assert` statements at the end on the program check that all the paths originating from the source node (represented by `0`) have positive weight.

```
int a[N];
void main(){
 int i;
 a={N-1,...,0};
 SelectSort();
 for(i=0;i<N;i++)
  assert(a[i]==i);
}
```

```
void SelectSort(){
 int i,j,t,min;
 for (j=0;j<N-1;j++){
  min=j;
  for (i=j+1; i<N; i++)
  if (a[i]>a[min])
   min = i;
  t = a[j];
  a[j] = a[min];
  a[min] = t;
 }
}
```

Fig. 7. Source code of `SelectSort.c`(N)



(a) Time (seconds) spent by the tools (b) Size of the formulae

Fig. 8. Results on `SelectSort.c`(N)

The results of the experiments are given in the plots of Fig. 10, where the x axis represents the number of edges. Notice that the maximum value for N is 20 in this case as the maximum number of edges in a fully connected directed graph is $k(k-1)$ where k is the number of nodes. Plot (a) displays the time spent by the tools in analysing the problems while plot (b) shows the size of the formulae. Notice that the formula generated by CBMC is already one order of magnitude bigger than the one of SMT-CBMC, for $N = 12$.

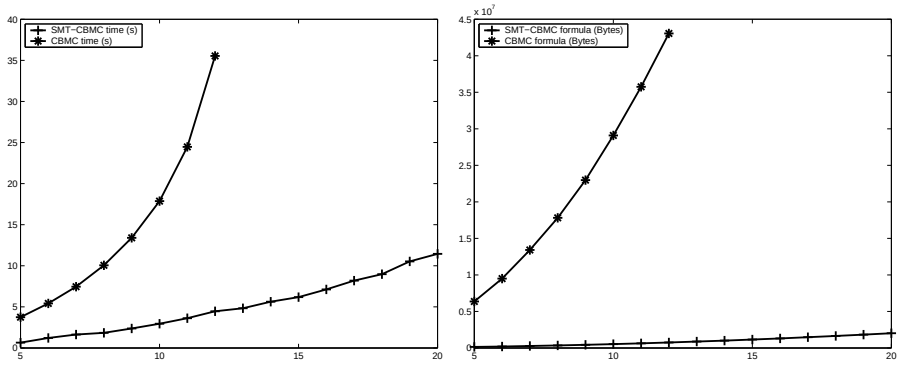
4.3 Prim's algorithm

Prim's algorithm [22] finds a minimum spanning tree for a connected weighted graph. As in the Bellman-Ford implementation (Fig. 9), three arrays are used to

```
int INFINITY = 899;
void main(){
  int nodecount = 5;
  int edgecount = 10;
  int source = 0;
  int Source[10] = {0,0,1,0,3,3,0,1,1,3};
  int Dest[10] = {1,1,1,1,2,4,4,2,3,3};
  int Weight[10] = {0,1,2,3,4,5,6,7,8,9};
  int distance[5];
  int x,y,i,j;

  for(i = 0; i < nodecount; i++){
    if(i == source) distance[i] = 0;
    else distance[i] = INFINITY;

  for(i = 0; i < nodecount; i++){
    for(j = 0; j < edgecount; j++){
      x = Dest[j];
      y = Source[j];
      if(distance[x] > distance[y] + Weight[j])
        distance[x] = distance[y] + Weight[j];
      }
  }

  for(i = 0; i < edgecount; i++){
    x = Dest[i];
    y = Source[i];
    if(distance[x] > distance[y] + Weight[i]) return;
  }

  for(i = 0; i < nodecount; i++) assert(distance[i]>=0);
}
```

Fig. 9. Source code of the instance of `BellmanFord`(N) for $N = 10$.

model the attributes of the edges that connect the nodes of the graph. We used instances where the number of nodes of the graph is set to 4 and the number of edges increases according to the parameter N, starting from $N = 4$. As shown in Table 1, already for $N = 4$ the size of the formula output by SMT-CBMC is roughly 37 times smaller than the one of CBMC. For $N = 7$ the difference becomes greater: the formula generated by SMT-CBMC becomes roughly 60 times smaller than the one of CBMC. The generation of compact formulae has an impact on the time spent by SMT-CBMC in resolving them: the formula generated for $N = 7$ is resolved in less than 20 seconds while CBMC takes roughly two minutes, with a difference of one order of magnitude.



(a) Time spent by the tools (b) Size of the formulae

Fig. 10. Results on `BellmanFord.c`(N)

Table 1. Performance of SMT-CBMC and CBMC against `Prim`(N).

	Size (Bytes)		Time (s)	
N	SMT-CBMC	CBMC	SMT-CBMC	CBMC
4	381,233	14,269,837	2.99	9.27
5	702,954	31,453,119	5.31	28.08
6	1,170,649	59,675,705	10.81	43.66
7	1,810,798	108,045,992	17.78	151.08

5 Related Work

In the recent years a number of verification procedures and tools have been developed for the analysis of software.

ESC/Java [24] is an effective error-detection tool for Java. User-annotated Java programs are analysed by generating formulae (called *verification conditions*) that represent those initial states from which no execution can violate the given properties. Such formulae are generated according to the semantics of weakest preconditions [25] and checked for validity with the Simplify theorem prover [26]. The use of weakest preconditions to build the verification condition imposes the computation of loop invariants. In order to ensure termination (finding loop invariants is an undecidable problem) several heuristics are employed, but this may lead to possible unsound outcomes of the analysis.

In order to extract error traces from counterexamples, in [27] the authors use *labelling functions* to build particular predicate symbols (labels) that are then added to the verification condition. The labels contain information that can be syntactically and automatically extracted from counterexamples in order to detect the exact position of an error in the source code. Our approach, by directly extracting error traces from counterexamples, does not clutter the formula fed to

the SMT solver with extra-logical information. On the other hand it requires the invocation of the SMT solver whenever a condition of a conditional expression is met during the transversal of the control flow graph. However in our experiments the time spent by our tool to carry out this activity is always negligible.

SLAM [28], BLAST [29], and MAGIC [30] extend a symbolic model checking procedure for boolean programs with abstraction and refinement. Their approach has been shown to be very effective on specific application domains such as device drivers programming. However, when they come to reason about arrays they trade precision for efficiency. For instance SLAM and BLAST do not distinguish different elements of an array and this may lead them to report unsound results.

Saturn [31] is an efficient software error-detection tool that, like CBMC, translates C programs into boolean formulae that are then fed to a SAT solver. One of the distinguishing features of Saturn w.r.t. CBMC is the computation of summaries for each analysed function in order to speed up the (interprocedural) analysis. But again efficiency is obtained at the cost of losing soundness: similarly to SLAM and BLAST, Saturn does not distinguish different elements of an array.

Both CBMC and SMT-CBMC treat arrays in a precise way, but they only consider execution traces of bounded length, limitation that can be mitigated by iterating the technique for increasing values of the unwinding bound. As shown in Section 4 SMT-CBMC can be considerably more effective than CBMC when applied to programs involving arrays of non-negligible size. However when no arrays occur in the program or when the arrays have small size CBMC can be more effective than SMT-CBMC. This suggests that the compilation to SMT should be seen as a complement and not as an alternative to the compilation to SAT. An interesting point is to determine syntactic criteria that allow us to determine for any given program which of the two encoding techniques is likely to perform best.

6 Conclusion

We have presented a Bounded Model Checking technique for sequential programs which uses SMT solvers instead of SAT solvers. Our work generalises the one presented in [2] and we have shown that our encoding technique generates considerably more compact formulae than CBMC when arrays are involved in the input program. In particular the size of the formulae generated by our approach does not depend on the size of the bit-vector representation of the basic data types nor on the size of the arrays occurring in the program.

Experimental results confirm the effectiveness of our approach: on problems involving complex interactions of arithmetics and arrays manipulation SMT-CBMC scales significantly better than CBMC as the size of the arrays occurring in the input program increases.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In Cleaveland, R., ed.: Proceedings of TACAS99. Volume 1579 of Lecture

Notes in Computer Science., Springer (1999) 193–207
2. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of DAC03, ACM Press (2003) 368–371
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Proceedings of TACAS04. Volume 2988 of LNCS., Springer (2004) 168–176
4. Nelson, G., Oppen, D.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
5. Shostak, R.E.: Deciding Combinations of Theories. Journal of ACM **31** (1984) 1–12
6. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In Etessami, K., Rajamani, S., eds.: 17th International Conference on Computer Aided Verification, Springer (2005) 20–23
7. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: Proceedings of LICS01, IEEE (2001)
8. Armando, A., Ranise, S., Rusinowitch, M.: A Rewriting Approach to Satisfiability Procedures. Information and Computation **183** (2003) 140–164
9. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisal. In Gramlich, B., ed.: Proceedings of FroCoS05. Volume 3717 of Lecture Notes in Computer Science., Springer (2005) 65–80
10. Cyrluk, D., Möller, O., Rueß, H.: An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In: 9th International Conference on Computer Aided Verification (CAV97). Volume 1254 of Lecture Notes in Computer Science., Springer (1997) 60–71
11. Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Proceedings of DAC98. (1998) 522–527
12. Grumberg, O., ed.: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. In Grumberg, O., ed.: CAV. Volume 1254 of Lecture Notes in Computer Science., Springer (1997)
13. Möller, O., Rueß, H.: Solving Bit-Vector Equations. In Gopalakrishnan, G., Windley, P., eds.: Formal Methods in Computer-Aided Design (FMCAD '98). Volume 1522 of Lecture Notes in Computer Science., Palo Alto, CA, Springer-Verlag (1998) 36–48
14. Bozzano, M., Bruttomesso, R., Cimatti, A., Franzen, A., Hanna, Z., Khasidashvili, Z., Palti, A., Sebastiani, R.: Encoding RTL Constructs for MATHSAT: a Preliminary Report. In Armando, A., Cimatti, A., eds.: 3rd International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005). Volume 144 of Electronic Notes in Theoretical Computer Science. (2005)
15. Clarke, E., Kroening, D., Yorav, K.: Behavioral Consistency of C and Verilog Programs. Technical Report CMU-CS-03-126, Computer Science Department, School of Computer Science, Carnegie Mellon University (2003)
16. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA (1986)
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC01, ACM (2001) 530–535
18. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In Alur, R., Peled, D., eds.: Proceedings of CAV04. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 515–518

19. Knuth, D.: The Art of Computer Programming, Volume 3: Sorting and Searching. Volume 3. Addison-Wesley (1997)
20. Bellman, R.E.: On a Routing Problem. Quarterly of applied mathematics **16** (1958) 87–90
21. Ford, L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press (1962)
22. Prim, R.C.: Shortest Connection Networks and Some Generalisations. Bell System Technical Journal **36** (1957) 1389–1401
23. Stump, A., Barrett, C.W., Dill, D.L.: CVC: a Cooperating Validity Checker. In Brinksma, E., Larsen, K.G., eds.: Proceedings of CAV02. Volume 2404 of LNCS., Springer (2002)
24. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 234–245
25. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18** (1975) 453–457
26. Detlefs, D.L., Nelson, G., Saxe, J.B.: Simplify: a Theorem Prover for Program Checking. Technical Report 148, HP Labs (2003)
27. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. **55** (2005) 209–226
28. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Proceedings of SPIN, Springer New York, Inc. (2001) 103–122
29. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In Ball, T., Rajamani, S.K., eds.: Proceedings of SPIN03. Volume 2648 of LNCS., Springer (2003) 235–239
30. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. In: Proceedings of ICSE03, IEEE Computer Society (2003) 385–395
31. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In Palsberg, J., Abadi, M., eds.: Proceedings of POPL05, ACM Press (2005) 351–363