

Building SMT-based Software Model Checkers: an Experience Report

Alessandro Armando

`armando@dist.unige.it`

Artificial Intelligence Laboratory (AI-Lab)
DIST - University of Genova, Italy

Abstract. In this paper I report on my experience on developing two SMT-based software model checking techniques and show—through comparison with rival state-of-the-art software model checkers—that SMT solvers are key to the effectiveness and scalability of software model checking.

1 Introduction

Software model checking is one of the most promising techniques for automatic program analysis. This is witnessed by the growing attention that this technique is receiving by leading software industries which have already introduced in their software production cycle:

Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.

Bill Gates [Gat02]

As the name suggests, software model checking owns its origins to model checking [Cla00], a powerful technique capable to perform a complete and fully automatic exploration of the behaviors of a finite state system by using sophisticated data structures and algorithms (most notably OBDDs and SAT solvers).

Model checking has been (and still is) remarkably successful in the analysis of hardware and communication protocols. However the application of model checking to software is considerably more difficult as most programs deal with very large or even infinite data structures and are therefore inherently infinite-state. The development of software model checking tools asks for a technology that supports automated reasoning about data structures commonly occurring in programs (e.g. integers, lists, arrays, bit-vectors). This goes clearly beyond the scope of OBDDs and SAT solvers.

The development of decision procedures for theories of data structures is a long standing problem in Automated Reasoning. The Satisfiability Modulo Theory (SMT) problem is the problem of determining whether a formula is satisfied by at least one model of a given theory \mathcal{T} . Of course, theories whose SMT

problem is decidable are of special interest. State-of-the-art SMT-solvers provide decision procedures for a number of decidable theories used in program verification, including propositional logic, linear arithmetics, the theory of uninterpreted function symbols, the theory of arrays, the theory of records and the combination thereof. (See [Seb07] for a survey on SMT.)

In this paper I report on my experience in developing two SMT-based software model checking techniques. The first technique carries out a bounded analysis of the input program through a reduction to a SMT problem. The second technique is based on the CounterExample-Guided Abstraction Refinement (CEGAR) paradigm and heavily relies on SMT solving during the refinement phase. Both techniques have been implemented in prototype model checking tools. I also show—through experimental comparison with rival state-of-the-art software model checkers—that SMT solvers are key to the effectiveness and scalability of the tools.

Structure of the paper. Section 2 shows that bounded model checking of software can greatly benefit if SMT solvers are used instead of SAT Solvers. Section 3 introduces the CEGAR paradigm and shows the prominent role played by SMT-solvers in this context. A brief survey of the related work is given in Sect. 4. Some concluding remarks are drawn in Sect. 5.

2 Bounded Model Checking of Software

For simplicity in this paper we focus on fragments of the C programming language containing the usual control-flow constructs (e.g. `if`, `while`, and `assert`), (recursive) function definitions, and with assignments (over numeric variables and arrays) and function calls as atomic statements. Programs may also contain *conditional expressions*, i.e. expressions of the form $(c?e_1:e_2)$ whose value is the value of e_1 if the value of c is different from 0 and is the value of e_2 otherwise. We also assume that the failure of an `assert` statement leads the program to a special control location denoted by 0. If a is a program variable of type array, then $\text{size}(a)$ denotes its size.

The *bounded reachability problem for a program P* is the problem of determining whether there exists an execution path of P of at most a given length reaching a given control location of P. Here I focus on the *bounded 0-reachability problem for P*, i.e. the problem of determining whether there exists an execution path of P of bounded length reaching control location 0. In this section I show how to reduce this problem to a SMT problem. (For the sake of simplicity, in this section I assume that `=` is the only assignment operator occurring in P and that no pointer variables nor conditional expressions occur in P.)

Given an integer k , the procedure generates an SMT formula whose models (if any) correspond to execution traces (of length bounded by k) leading to an assertion violation. Finding assertion violations (of length up to k) therefore boils down to solving SMT problems.

Preliminarily to the generation of the formula, we apply a number of simplifying transformations to P, thereby obtaining a simplified program S, whose

execution paths correspond to finite prefixes of the execution paths of P . These transformations are described in Section 2.1. We then build two sets of quantifier-free formulae \mathcal{C}_S and \mathcal{P}_S such that

$$\mathcal{C}_S \models_{\mathcal{T}} \bigwedge \mathcal{P}_S \quad (1)$$

for some given background theory \mathcal{T} if and only if no execution path of S violates any **assert** statement. The generation of \mathcal{C}_S and \mathcal{P}_S is the described in Sect. 2.2. The usage of SAT and SMT solvers to tackle (1) is discussed in Sect. 2.3 and in Sect. 2.3 respectively.

2.1 The Preprocessing Phase

The preprocessing activity starts by unwinding loops, i.e. by replacing them with a sequence of nested **if** statements. This is done by removing **while** loops through the application of the following transformation k times:

$$\text{while}(e) \{ P' \} \longrightarrow \text{if}(d) \{ P' \text{ while}(e) \{ P' \} \}$$

and by replacing the remaining **while** loop with an *unwinding assertion* of the form **assert(!e)**; . The failure of an unwinding assertion indicates that the bound k is not sufficient to adequately model the problem at hand, thereby indicating that it must be increased. Non-recursive function calls are then inlined. Recursive function calls are unwound similarly to loop statements. Let Q be the program obtained from P by applying the above transformations. It is easy to see that the execution paths of Q correspond to finite prefixes of the execution paths of P .

Next, program Q is put in *Static Single Assignment* (SSA) form. Let R be the resulting program. A program in SSA form [CFR⁺89] is a program in which every variable is assigned at most once. The transformation in SSA form is done by

1. replacing all the assignments of the form $\mathbf{a}[e_1]=e_2$; with $\mathbf{a}=\text{store}(\mathbf{a}, e_1, e_2)$; , where **store** is a function such that $\text{store}(\mathbf{a}, e_1, e_2)$ returns the array obtained from \mathbf{a} by setting the element at position e_1 with the value of e_2 ;
2. replacing the occurrences of the variables that are target of assignments (say \mathbf{x}) with new, indexed versions $\mathbf{x}_0, \mathbf{x}_1, \dots$;
3. replacing all the occurrences of the variables that are not target of assignments with appropriate versions so to preserve the semantics of the original program; and
4. adding a new assignment of the form $\mathbf{x}_{j_3}=(c?\mathbf{x}_{j_1}:\mathbf{x}_{j_2})$; (for suitable values of j_1, j_2 , and j_3) after each conditional statement of the form **if**(c) Q_1 [**else** Q_2] where \mathbf{x} occurs as target of an assignment in Q_1 [or in Q_2].

An example of transformation in SSA form is given in Fig. 1. As the first two statements of Q have the same target variable (namely \mathbf{i}), the target of the corresponding assignments in R are two distinct versions of the same variable (namely \mathbf{i}_1 and \mathbf{i}_2). The assignment at line 3 of R uses in its right-hand side

the version of the variable that is target of the assignment at line 2, namely i_1 . The same considerations apply to the two occurrences of x that are target of the assignments at lines 6 and 8. Notice that the additional assignments at lines 9 and 11 of R are added to provide a unique definition for the future uses of x .

<pre> 1 // Program Q 2 i=a[0]; 3 i=i+1; 4 if(x>0) { 5 if(x<10) 6 x=x+1; 7 else 8 x=x-1; 9 } 10 assert(x>0 && x<10); 11 a[x]=i; </pre>	<pre> 1 // Program R 2 i₁=a₀[0]; 3 i₂=i₁+1; 4 if(x₀>0) { 5 if(x₀<10) 6 x₁=x₀+1; 7 else 8 x₂=x₀-1; 9 x₃=(x₀<10?x₁:x₂); 10 } 11 x₄=(x₀>0?x₃:x₀); 12 assert(x₄>0 && x₄<10); 13 a₁=store(a₀,x₄,i₂); </pre>
---	--

Fig. 1. Example program Q (left) and corresponding program in SSA form R (right)

Notice that by turning a program into SSA form we are trading assignments (e.g. $x=x+1$;) for equalities (e.g. $x_1 = x_0 + 1$). This is why, preliminarily to the generation of the encoding, it is convenient to apply this transformation.

The program R is now turned in *Conditional Normal form*, i.e. into a sequence of statements of the form $\text{if}(c) r$, where r is either an assignment or an assertion and does not contain conditional expressions. We refer to statements of the form $\text{if}(c) r$ as *conditional statements*. Notice that this normalization step removes the **else** constructs and pushes the **if** statements downwards in the abstract syntax tree of the program until they are applied to atomic statements only.

The program in Conditional Normal form S corresponding to the program R on the right of Fig. 1 is shown on the left of Fig. 2. As the execution of statements at lines 2, 3, 12 and 13 of R does not depend on any condition, the guard of the corresponding conditional statements in S is **true**. The assignment at line 6 of R is executed only if the conditions of the two preceding **if** statements hold. Therefore the corresponding assignment at line 4 in S is guarded by the conjunction of these two conditions. Similar considerations apply for the guard of the assignment at line 5 in S. The assignment at line 9 (11) of R is turned into the pair of conditional statements at lines 6 and 7 (8 and 9, resp.) of S.

It must be noted that S is not necessarily in SSA form. However, all the variables that are assigned more than once (e.g. x_3 and x_4 in the program of Fig. 2) are guarded by mutually exclusive conditions.

Notice that, for suitable values of k , all the above transformations (i.e. the transformations leading from the input program P to Q, from Q to R, and from

<pre> 1 // Program S 2 if (true) i₁=a₀[0]; 3 if (true) i₂=i₁+1; 4 if (x₀>0 && x₀<10) x₁=x₀+1; 5 if (x₀>0 && !(x₀<10)) x₂=x₀-1; 6 if (x₀>0 && x₀<10) x₃=x₁; 7 if (x₀>0 && !(x₀<10)) x₃=x₂; 8 if (x₀>0) x₄=x₃; 9 if (!(x₀>0)) x₄=x₀; 10 if (true) assert(x₄>0 && x₄<10); 11 if (true) a₁=store(a₀, x₄, i₂); </pre>	$ \begin{aligned} \mathcal{C}_S = \{ & (\text{TRUE} \Rightarrow i_1 = \text{select}(a_0, 0)), \\ & (\text{TRUE} \Rightarrow i_2 = i_1 + 1), \\ & ((x_0 > 0 \wedge x_0 < 10) \Rightarrow x_1 = x_0 + 1), \\ & ((x_0 > 0 \wedge \neg(x_0 < 10)) \Rightarrow x_2 = x_0 - 1), \\ & ((x_0 > 0 \wedge x_0 < 10) \Rightarrow x_3 = x_1), \\ & ((x_0 > 0 \wedge \neg(x_0 < 10)) \Rightarrow x_3 = x_2), \\ & (x_0 > 0 \Rightarrow x_4 = x_3), \\ & (\neg(x_0 > 0) \Rightarrow x_4 = x_0), \\ & (\text{TRUE} \Rightarrow a_1 = \text{store}(a_0, x_4, i_2)) \} \\ \mathcal{P}_S = \{ & \text{TRUE} \Rightarrow (x_4 > 0 \wedge x_4 < 10) \} \end{aligned} $
--	--

Fig. 2. Program in Conditional Normal S (left) and corresponding encoding (right)

R to the program S) are such that each execution path in the input program corresponds to an execution path in the output program and vice versa, and both paths contain the same (modulo renaming of the variables) sequence of atomic statements, and all atomic statements are guarded by the same (modulo renaming of the variables) conditions. From this fact it readily follows that the bounded reachability problem for P can be reduced to the reachability problem for S.

2.2 The Encoding Phase

Let $S = s_1 \cdots s_m$ be the program in Conditional Normal form resulting from the application of the transformations described in Section 2.1 to the input program P for a given value of k . We now show how to build two sets of quantifier-free formulae \mathcal{C}_S and \mathcal{P}_S such that $\mathcal{C}_S \models_{\mathcal{T}} \bigwedge \mathcal{P}_S$ if and only if no execution path of S violates any `assert` statement.

For simplicity we assume that the variables of S are either of type `int` or are arrays of elements of type `int`. We define \mathcal{T} to be the union of a theory of the integers and the theory of arrays. We also assume that the language of \mathcal{T} contains (i) a variable v_j of sort `INT` for each variable v_j of S of type `int` and (ii) a variable a_j of sort `array` for each variable a_j of S ranging over arrays. Let e be a program expression. By e^* we indicate the term of the language of \mathcal{T} obtained from e by replacing all program variables (say v_j) with the corresponding variables of \mathcal{T} (say v_j) and the operators occurring in e with the corresponding function and predicate symbols in \mathcal{T} . For instance, if $e = (a_1[v_1+1] \leq v_0+2)$, then $e^* = (\text{select}(a_1, v_1 + 1) \leq v_0 + 2)$.

For each statement in S of the form `if(c) vj=e;`, \mathcal{C}_S contains the formula $(c^* \Rightarrow (v_j = e^*))$ and for each statement of the form `if(c) assert(e);` in S, \mathcal{P}_S contains the formula $(c^* \Rightarrow e^*)$. An example of the encoding is given in Fig. 2.

If S is a program in conditional normal form with m statements, then $\mathcal{C}_S \models_{\mathcal{T}} \bigwedge \mathcal{P}_S$ if and only if all complete execution paths of S end in control location $m + 1$. This result (proved in [AMP09]) guarantees that the encoding given by \mathcal{C}_S and \mathcal{P}_S

is sound and complete, i.e. the bounded 0-reachability problem can be reduced to a SMT problem.

2.3 The Solving Phase

Solving the Formulae with a SAT Solver. In [KCY03] checking problems of the form (1) is reduced to a propositional satisfiability problem which is then fed to a SAT solver. This is done by modeling variables of basic data types (e.g. `int`) as fixed-size bit-vectors and by considering the equations in \mathcal{C}_S and in \mathcal{P}_S as bit-vector equations. Each array variable a is also replaced by $\text{size}(a)$ distinct variables $a^0, \dots, a^{\text{size}(a)-1}$ and each formula of the form $c \Rightarrow (a_{j+1} = \text{store}(a_j, e_1, e_2))$ occurring in \mathcal{C}_S is replaced by the formula $\bigwedge_{i=0}^{\text{size}(a)-1} a_{j+1}^i = ((c \wedge e_1 = i) ? e_2 : a_j^i)$, where $v = (c ? e_1 : e_2)$ abbreviates the formula $(c \Rightarrow v = e_1) \wedge (\neg c \Rightarrow v = e_2)$. Finally each term of the form $\text{select}(a_j, e)$ is replaced by a new variable, say x , and $\bigwedge_{i=0}^{\text{size}(a)-1} ((e = i) \Rightarrow x = a_j^i)$ is added to \mathcal{C}_S .

The resulting set of bit-vector equations is then turned into a propositional formula. 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. More generally, if the program contains a multi-dimensional array a with dimensions d_1, \dots, d_m , then the number of added formulae grows as $O(d_1 \cdot d_2 \cdot \dots \cdot d_m)$.

Solving the Formulae with a SMT Solver. The alternative approach, first proposed in [AMP06], is to use a SMT solver to directly check whether $\mathcal{C}_S \models_{\tau} \bigwedge \mathcal{P}_S$. 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. Moreover the use of a SMT solver gives additional freedom in the modeling the basic data types. In fact, program variables of numeric type (e.g. `int`, `float`) can be modeled by variables ranging over bit-vectors or over the corresponding numerical domain (e.g. \mathbb{Z} , \mathbb{R} , resp.). If the modeling 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 modeling of numeric variables is done through the corresponding numerical domain, then the result of the analysis is independent from the actual binary representation.

2.4 Experimental Results

SMT-CBMC is an SMT-based bounded model checker for sequential programs that implements the ideas described in Sect. 2. SMT-CBMC consists of four main modules. 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 by invoking CVC Lite [BB04] as a SMT solver. The latter module also builds and prints the error trace whenever a counterexample is returned by CVC Lite. SMT-CBMC can represent numeric data types with

corresponding numeric domains as well as with fixed-size bit-vectors. Moreover the user can specify the maximum number of unwindings to be considered.

We have thoroughly assessed the approach by running SMT-CBMC and CBMC against the following families of C programs:

- `SelectSort.c(N)`, an implementation of the Selection Sort algorithm [Knu97],
- `BellmanFord.c(N)`, an implementation of the Bellman Ford algorithm [Bel58] for computing single-source shortest paths in a weighted graph, and
- `m_k_Gray_codes.c(N)`, an implementation of an algorithm for the generation of (m, k) -Gray code [Bla05], a generalization of the binary Gray code.

Each family of programs is parametric in a positive integer N 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 results of the experiments are reported in Fig. 3. All the experiments presented here and in the rest of this paper have been carried out on a 2.4GHz Pentium IV running Linux with memory limit set to 800MB and time limit set to 30 minutes. The time (in seconds) spent by the tools to tackle each individual instance is given in the plots on the left. The size of the encodings (in bytes) generated by the tools are given by the plots on the right. CBMC has been invoked by manually setting the unwinding bound and by disabling simplification as these functionalities are not available in SMT-CBMC.

On the instances in the `SelectSort.c(N)` family, CBMC runs out of memory for $N > 17$, while SMT-CBMC can still analyze programs for $N = 75$. A comparison between the size of the formulae generated by SMT-CBMC and CBMC substantiates our remarks about the size of the encodings: the formula built by SMT-CBMC for $N = 17$ is almost two orders of magnitude smaller than the one built by CBMC. Similar considerations apply on the results obtained by running the tools against the instances of the `BellmanFord.c(N)` and `m_k_Gray_Code.c(N)` families.

It is worth pointing out that in our experiments we modeled the basic data types with bit-vectors thereby exploiting 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 letting the numerical variables range over the integers and thereby using the decision procedure for linear arithmetic available in CVC Lite during the solving phase.

3 Counterexample-Guided Abstraction Refinement

Given a program P as input, a model checking procedure based on CEGAR amounts to the iteration of the following steps (see also Fig. 4):

Abstraction. An abstract program \hat{P} is generated from P . By construction every execution trace of P is also an execution trace of \hat{P} . However, some trace of \hat{P} may not correspond to a trace of P .

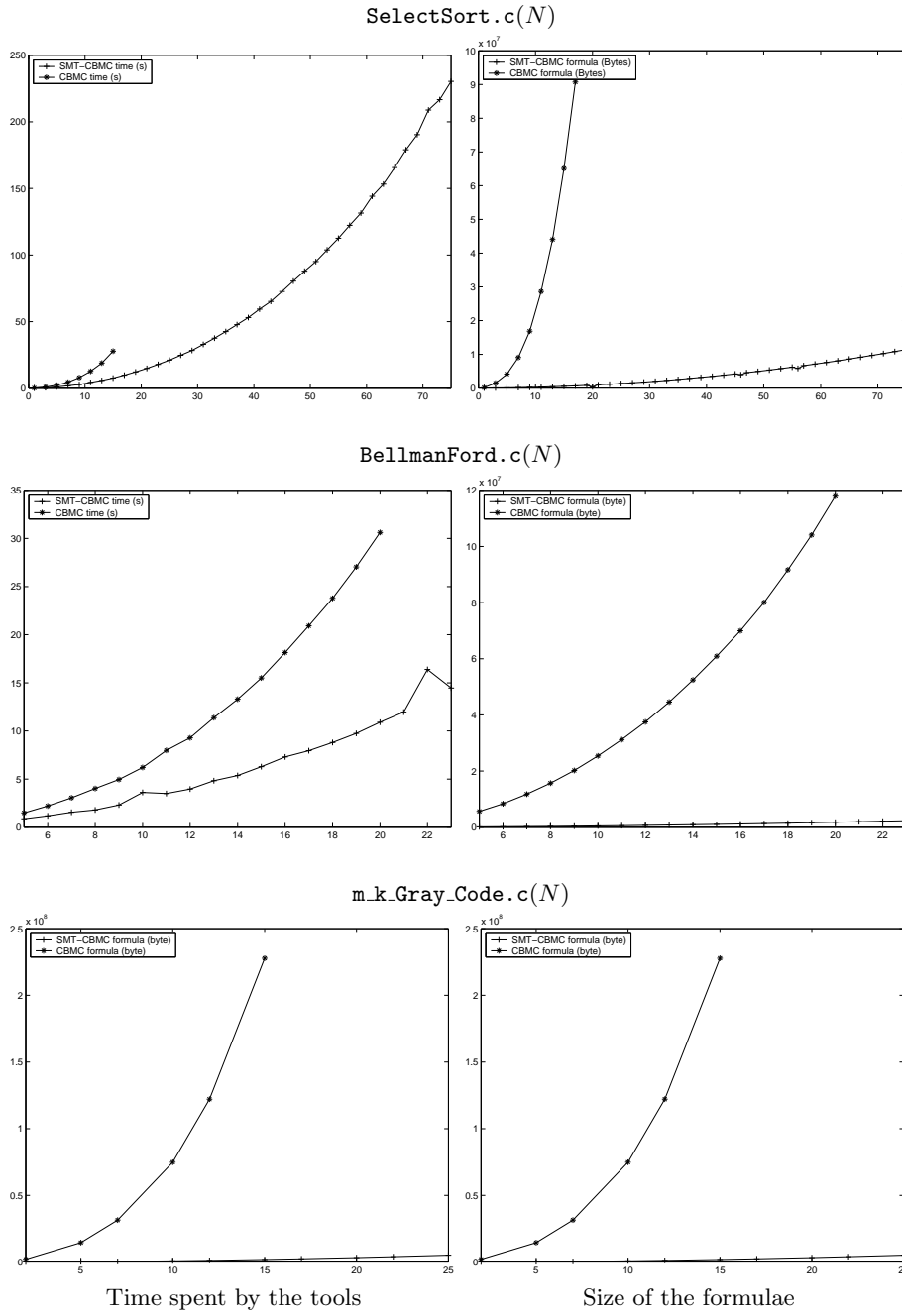


Fig. 3. SMT-CBMC vs. CBMC: time spent by the tools (left) and size of the encodings (right)

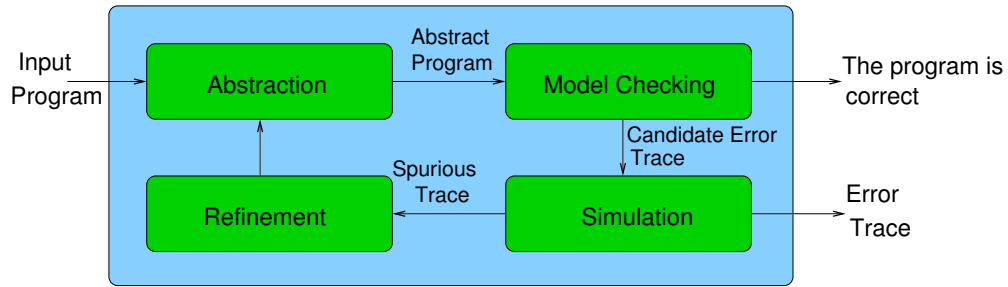


Fig. 4. The CEGAR Loop

Model Checking. The abstract program \hat{P} is model-checked. If \hat{P} is found to be safe (i.e. correct w.r.t. given properties), so is the original program P and this is reported to the user. If an error trace is found in \hat{P} , then it is given as input to the next step.

Simulation. The error trace of \hat{P} found in the previous step is *simulated* on P in order to determine its feasibility, i.e. if it corresponds to an error trace of P . The feasibility check is usually carried out with the aid of an SMT solver. If the trace is feasible, then the error trace is reported to the user, otherwise the procedure continues with the next step.

Refinement. The information gathered from the SMT solver during the simulation step (typically a proof of the unfeasibility of the trace) is used to generate a new abstract program which (i) is more precise than the previous abstraction, and (ii) does not contain the spurious error trace found by the model checking phase.

The procedure outlined above can be instantiated in a variety of ways depending on the abstraction chosen. This choice has an important impact on the effectiveness of the whole procedure as it affects the complexity of the component steps (namely how difficult is (i) to model check abstract programs, (ii) to check abstract error traces, and (iii) to compute refined versions of the current abstract program whenever a spurious counterexample is found) as well as the number of iterations needed by the procedure to terminate.

3.1 Boolean Programs and CEGAR with Predicate Abstraction

The most widely applied instance of CEGAR (e.g. SLAM [BR01], BLAST [HJMS02], SATABS [CKSY05]) is based on predicate abstraction and uses boolean programs as target of the abstraction, i.e. programs whose variables range over the booleans and model the truth values of predicates corresponding to properties of the program state. The nice feature of boolean programs is that their reachability problem is decidable. A model checker for boolean programs based on the OBDDs technology is described in [BR00].

To illustrate, consider the programs in Fig. 5. These programs are obtained by applying a CEGAR procedure based on predicate abstraction to program

P . Program \widehat{P}_0 is the first abstraction of P and is obtained by replacing all

<pre> 1 /* Program P */ 2 int numUnits; 3 int level; 4 5 void getUnit() { 6 int canEnter=F; 7 if(numUnits==0) { 8 if(level>0) { 9 NewUnit(); 10 numUnits=1; 11 canEnter=1; 12 } 13 } else canEnter=1; 14 if(canEnter==1) { 15 if(numUnits==0) 16 ERROR: ; 17 else 18 gotUnit(); 19 } 20 }</pre>	<pre> /* Program P-hat_0 */ void getUnit() { ; if(⊥) { if(⊥) { ; ; ; ; } else ; if(⊥) { if(⊥) ERROR: ; else ; } }</pre>	<pre> /* Program P-hat_1 */ bool nU0; void getUnit() { ; if(nU0) { if(⊥) { ; nU0=F; ; } else ; if(⊥) { if(nU0) ERROR: ; else ; } }</pre>	<pre> /* Program P-hat_2 */ bool nU0; void getUnit() { bool cE=F; if(nU0) { if(⊥) { ; nU0=F; cE=T; } else cE=T; if(cE) { if(nU0) ERROR: ; else ; } }</pre>	<pre> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</pre>
--	---	--	--	--

Fig. 5. Application of CEGAR with Predicate Abstraction

the atomic statements by the skip statement (;) and all the expressions occurring in conditionals statements and in assertions with the \perp symbol (standing for *undefined*). A model checker for boolean programs determines that line 16 reachable in \widehat{P}_0 and returns 6,7,14,15,16 as execution trace. This trace is found to be spurious by simulating its execution on P . In fact, `numUnits==0` cannot be false at line 7 and true at line 15. Program \widehat{P}_0 is thus refined into a program \widehat{P}_1 containing a new boolean variable `nU0` that is true when `numUnits==0` and false otherwise. The model checker now determines that line 16 is still reachable in \widehat{P}_1 with error trace 6,7,8,14,15,16. Simulation on P reveals that also this trace is spurious. In fact, the assignment at line 6 sets `canEnter` to 0 and hence `canEnter==1` at line 13 cannot possibly be true. Thus \widehat{P}_1 is refined to \widehat{P}_2 which contains a new boolean variable `cE` that is true when `canEnter==1` and false otherwise. The model checker now concludes that line 16 is not reachable in \widehat{P}_2 and the CEGAR procedure can thus halt and return that line 16 is not reachable in P as well.

Predicate abstraction has proved effective on important application domains such as operating systems device drivers [BBC⁺06]. But it must be noted that device drivers are inherently control-intensive (i.e. the control flow largely prevails over the data flow). The effectiveness of predicate abstraction refinement on other kinds of software is still to be ascertained. As a matter of fact, CEGAR based on predicate abstraction performs poorly when applied to data intensive programs (e.g. programs performing non trivial manipulations of numeric variables and/or arrays): the abstractions that are built are too coarse, too many iterations of the abstraction refinement loop are usually required, and the re-

finement phase often fails to determine the boolean variables needed to build a more accurate model.

3.2 Linear Programs and CEGAR with Array Abstraction

Linear Programs (with arrays) are C programs where variables (and array elements) range over a numeric domain and expressions involve linear combinations of variables and array elements. By *Linear Arithmetic* we mean standard arithmetic (over \mathbb{R} or \mathbb{Z}) with addition (i.e. +) and the usual relational operators (e.g. =, <, ≤, >, ≥) but without multiplication. By the *theory of arrays* we mean the theory concisely presented by the following axiom:

$$\forall a, i, j, e. \text{select}(\text{store}(a, i, e), j) = (j = i ? e : \text{select}(a, j)) \quad (2)$$

where $(_ ? _ : _)$ is a conditional term constructor with the obvious semantics.

An CEGAR procedure for model checking linear programs with arrays was is forward in [ABM07]. At the core of the procedure lies the model checker for linear programs described in [ACM04] and uses array indexes instead of predicates for the abstraction: the input program is abstracted w.r.t. sets of array indexes, the abstraction is a linear program (without arrays), and refinement looks for new array indexes. Thus, while predicate abstraction uses boolean programs as the target of the abstraction, in this approach linear programs are used for the same purpose. This is particularly attractive as linear programs can directly and concisely represent complex correlations among program variables and a small number of iterations of the abstraction refinement loop usually suffice to either prove or disprove that the input program enjoys the desired properties.

<pre> 1 /* Program P */ 2 void main() { 3 int i, a[30]; 4 a[1] = 9; 5 i = 0; 6 while(a[i]!=9) { 7 a[i] = 2*i; 8 i = i+1; } 9 assert(i<=1); }</pre>	<pre> 1 /* Program P̂₀ */ 2 void main() { 3 int i; 4 ; 5 i = 0; 6 while(⊥!=9) { 7 ; 8 i = i+1; } 9 assert(i<=1); }</pre>	<pre> 1 /* Program P̂₁ */ 2 void main() { 3 int i, a¹; 4 a¹ = (i==1)?9:a¹; 5 i = 0; 6 while(((i==1)?a¹:⊥)!=9) { 7 a¹ = (i==1)?2*i:a¹; 8 i = i+1; } 9 assert(i<=1); }</pre>
---	---	---

Fig. 6. Application of CEGAR with Array Abstraction

To illustrate, consider the linear program with arrays P in Fig. 6. The procedure starts by abstracting P into \hat{P}_0 . This done by replacing every occurrence of array expressions with the symbol \perp (denoting an arbitrary number) and every assignment to array elements with a skip statement (;). By applying a model checker for linear programs to \hat{P}_0 we get the execution trace 4, 5, 6, 7, 8, 6, 7, 8, 6, 9, 0 witnessing the violation of the assertion at line 9. The feasibility check of the above trace w.r.t. P is done by generating a set of quantifier-free formulæ

whose satisfying valuations correspond to all possible executions of the sequence of statements of P corresponding to the trace under consideration. This is done by first putting the trace in SSA form, then by replacing **while** statements with **assume** statements as shown in Tab. 1. Finally, quantifier-free formulæ encoding the behavior of the statements are generated.

Table 1. Checking the trace for feasibility.

Step	Line	Original Statement	Renamed Statement	Formula
1	4	a [1]=9;	a ₁ [1]=9;	$a_1 = \text{store}(a_0, 1, 9)$
2	5	i =0;	i ₁ =0;	$i_1 = 0$
3	6	while (a [i] != 9);	assume (a ₁ [i ₁] != 9);	$\text{select}(a_1, i_1) \neq 9$
4	7	a [i] = 2 * i ;	a ₂ [i ₁] = 2 * i ₁ ;	$a_2 = \text{store}(a_1, i_1, 2 * i_1)$
5	8	i = i + 1;	i ₂ = i ₁ + 1;	$i_2 = i_1 + 1$
6	6	while (a [i] != 9);	assume (a ₂ [i ₂] != 9);	$\text{select}(a_2, i_2) \neq 9$
7	7	a [i] = 2 * i ;	a ₃ [i ₂] = 2 * i ₂ ;	$a_3 = \text{store}(a_2, i_2, 2 * i_2)$
8	8	i = i + 1;	i ₃ = i ₂ + 1;	$i_3 = i_2 + 1$
9	6	while (a [i] != 9);	assume (!(a ₃ [i ₃] != 9));	$\neg(\text{select}(a_3, i_3) \neq 9)$
10	8	assert (i <= 1);	assert (i ₃ <= 1);	$\neg(i_3 \leq 1)$

The resulting set of formulæ is then fed to an SMT solver. If it is found unsatisfiable (w.r.t. the union of the theory of arrays and the theory of bit-vectors), then the trace is not executable in P . If it is found satisfiable, then we can conclude that the trace is also executable in P . In our example the set of formulæ (see rightmost column in Table 1) is unsatisfiable and the proof of unsatisfiability found by the SMT solver is the following:

$$\begin{array}{c}
\frac{Q(i_2, a) \vdash \text{select}(a_2, i_2) \neq 9 \vdash i_2 = i_1 + 1}{\frac{Q(i_1 + 1, a) \vdash \text{select}(a_2, i_1 + 1) \neq 9 \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1 + 1, a) \vdash \text{select}(\text{store}(a_1, i_1, 2 * i_1), i_1 + 1) \neq 9} \quad (2)}{Q(i_1 + 1, a) \vdash (i_1 + 1 = i_1 ? 2 * i_1 : \text{select}(a_1, i_1 + 1)) \neq 9} \\
\frac{Q(i_1 + 1, a) \vdash \text{select}(a_1, i_1 + 1) \neq 9 \vdash i_1 = 0}{Q(1, a) \vdash \text{select}(a_1, 1) \neq 9 \vdash a_1 = \text{store}(a_0, 1, 9)} \\
\frac{Q(1, a) \vdash \text{select}(\text{store}(a_0, 1, 9), 1) \neq 9}{Q(1, a) \vdash (1 = 1 ? 9 : \text{select}(a_0, 1)) \neq 9} \quad (2) \\
\frac{Q(1, a) \vdash \perp}{}
\end{array}$$

It is easy to see that the formulæ contributing to the proof are those associated with steps 1, 2, 4, 5, and 6. Moreover, the only term of the form $\text{select}(a, e)$ occurring in these formulæ is $\text{select}(a_2, i_2)$ (with $i_2 = 1$ given by the context). Thus, in order to rule out the above trace, we must refine \hat{P}_0 by including the element of **a** at position 1. The resulting program, \hat{P}_1 , is obtained by replacing every expression of the form $\mathbf{a}[e]$ with the conditional expression $((e==1)?\mathbf{a}^1:\mathcal{U})$ and every assignment of the form $\mathbf{a}[e_1]=e_2$; with the assignment $\mathbf{a}^1=((e_1==1)?e_2:\mathbf{a}^1)$; where \mathbf{a}^1 is a new variable of numeric type corresponding to the array element of index 1. The model checker can now conclude that node 0 cannot be reached in \hat{P}_1 and from this it follows that it is not reachable in P as well.

3.3 Experimental Results

The procedure presented in Sect. 3.2 has been implemented in the EUREKA tool [ABC⁺07]. We have tested EUREKA against a number of programs that involve reasoning on both arithmetic and arrays and thus allow to thoroughly assess the effectiveness and the scalability properties of the tool. On the same problems we have tested two well-known software model checkers based on predicate abstraction, namely BLAST and SATABS. All problems consist of a family of programs parametric in a positive integer N . The size of the arrays occurring in the programs and/or the number of iterations carried out by the loops increase as N increases. Thus the higher is the value of N , the bigger is the search space to be analyzed.

Table 2. Experimental results.

Safe instances							
Benchmark	EUREKA			BLAST	SATABS		
	N	total time	refined/array elems	N	total time	N	ref. time total time
STRING COPY	1000*	127.89	1/2 N	Incorrect	10	105.98	144.69
STRING COMPARE	1000*	28.62	2/2 N	Incorrect	9	210.619	296.20
GRAY CODE	60	62.75	16/28	Incorrect	Inconclusive		
PARTITION	40	108.23	1/ N	Incorrect	Inconclusive		
BUBBLE SORT	9	77.67	N/N	Incorrect	2	24.39	30.42
INSERTION SORT	16	60.86	N/N	Incorrect	2	51.43	74.74
SELECTION SORT	9	64.20	N/N	Incorrect	2	75.53	115.86

Unsafe Instances							
Benchmark	EUREKA			BLAST	SATABS		
	N	total time	refined/array elems	N	total time	N	ref. time total time
STRING COPY	1000*	73.70	0/2 N	100*	167.98	21	724.40 819.77
STRING COMPARE	1000*	15.07	0/2 N	100*	435.26	12	292.19 348.19
GRAY CODE	1000*	1.70	12/28	1000*	7.62	Inconclusive	
PARTITION	1000*	61.42	0/ N	10*	214.31	21	127.91 186.63
BUBBLE SORT	50*	26.19	0/ N	15*	395.38	7	327.33 460.76
INSERTION SORT	100*	190.74	0/ N	20*	316.00	5	131.58 270.64
SELECTION SORT	500*	41.15	0/ N	20*	506.81	7	101.97 291.20

The results in Table 2 refers to safe and unsafe (i.e. with a bug injected) instances of the above benchmarks. Each entry shows the greatest instance the tools are able to analyze and the time in seconds. Also, we give the time taken by the refinement phase of SATABS, the number of array elements found by EUREKA during the refinement phase, and the sum of the sizes of the arrays involved in the programs. Numbers with * indicate that the tool can analyze greater instances than the one shown.

The analysis of linear programs with arrays is obtained by letting each tool deal with arrays in its own way: EUREKA abstracts and refines them w.r.t. array indexes, SATABS models them faithfully as adjacent sequences of variables, and BLAST applies some abstraction techniques. (Expressions of the form $a[i]$ and $a[i+r]$, as well as $*(a+i)$ and $*(a+i+r)$, with $r > 0$ are indistinguishable for BLAST [HJMS02].)

As shown in Table 2, on most safe problems BLAST reports an incorrect answer, that is, it concludes that the program is unsafe when it is safe instead. On unsafe instances BLAST exhibits a better behavior than on safe ones, but still it exhibits scalability problems.

SATABS employs a SAT solver in the abstraction and refinement phases, thus allowing a precise encoding of pointer arithmetic, bit-level constructs, etc. Table 2 reveals (as expected) that SATABS does not output false positives nor negatives, and that scalability is often an issue. The results of the experiments demonstrate the great effort required by the refinement phase despite the efficiency of current SAT solvers.

The experimental results confirm the effectiveness of the array abstraction and reveal the difficulties of the approaches based on predicate abstraction when dealing with programs featuring a tight interplay between arithmetic and array manipulation.

4 Related Work

Bultan, Gerber, and Pugh are probably among the first to investigate model checking of infinite-state systems. In [BGP99] they propose a model checking technique for concurrent systems where transition systems are specified using Presburger formulæ over the integers. The systems are formalized in an *ad-hoc* event-action input language which no provision for arrays. Termination of the procedure process is ensured by means of widening techniques [CC77].

Besides BLAST and SATABS, which we analyzed in the previous sections, a number of other tools based on predicate abstraction have been developed. The early SLAM project¹ has now given rise to the Static Driver Verifier tool².

F-Soft [ISGG05] is a model checker for C programs developed at NEC Labs America. F-Soft treats pointers and arrays precisely, but this is done by fully expanding arrays and using an explicit representation of the internal memory. Since F-Soft is not publicly available no experimental comparison has been possible.

The approach proposed in [CR06] reduces the problem of finding execution paths of finite length that violate some given properties to a Constraint Satisfaction Problem (CSP). This is done by building a boolean constraint system whose solutions correspond to the execution paths of the control-flow graph of the program. A SAT solver is then used to enumerate the execution paths. For each execution path found by the SAT solver a constraint system encoding the reachability of an error statement is built and fed to a constraint solver for finite

¹ See <http://www.research.microsoft.com/~slam>.

² See <http://www.microsoft.com/whdc/devtools/tools/sdv.msp>.

domains. The approach is thus clearly exponential in the number of execution paths of the control-flow graph of the program.

Flanagan [Fla04] provides an in-depth analysis of the connection between Constraint Logic Programs (CLPs), imperative, and object oriented programs. CLPs are embedded in the CEGAR paradigm in order to combine and unify theorem proving (performed with CLP over a numeric domain) and model checking of boolean programs (performed using CLP over the booleans). However, to the best of our knowledge, no implementation is publicly available.

ESC/Java [FLL⁺02] analyzes user-annotated Java programs by generating verification conditions (representing the initial states from which no execution can lead to a violation of the given properties) which are then checked with the Simplify theorem prover [DNS03]. Since the generation of the verification conditions is an undecidable problem, several heuristics are used to drive this activity, but this may lead the tool to report unsound results.

5 Conclusions

SMT solvers are a key technology for the development of effective software model checkers. In this paper I have shown that by leveraging the strengths of SMT solvers it is possible to build software model checkers that outperform state-of-the-art tools based on OBDDs and SAT solvers.

Acknowledgments

I am indebted to Jacopo Mantovani, Lorenzo Platania, and Massimo Benerecetti for their collaboration on the development of the software model checking techniques presented in the paper. Most of the work I have reported here would not have been possible without their contribution. I am also grateful to Dario Carotenuto and Pasquale Spica for their help in the development of the EUREKA tool.

This work was partially supported by the PRIN Project no. 20079E5KM8 “Integrating automated reasoning in model checking: towards push-button formal verification of large-scale and infinite-state systems” funded by the Italian Ministry of University and Research.

References

- [ABC⁺07] Alessandro Armando, Massimo Benerecetti, Dario Carotenuto, Jacopo Mantovani, and Pasquale Spica. The EUREKA tool for software model checking. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 541–542. ACM, 2007.
- [ABM07] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Abstraction refinement of linear programs with arrays. In *TACAS*, volume 4424 of *LNCS*, Braga, Portugal, 2007. Springer.

- [ACM04] Alessandro Armando, Claudio Castellini, and Jacopo Mantovani. Software model checking using linear constraints. In *ICFEM'04*, volume 3308 of *LNCS*, Seattle, USA, 2004. Springer.
- [AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers. In *SPIN*, volume 3925 of *LNCS*, Vienna, Austria, 2006. Springer.
- [AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):69–83, 2009.
- [BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, volume 3114, pages 515–518, Boston, 2004. Springer.
- [BBC⁺06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 73–85, New York, NY, USA, 2006. ACM Press.
- [Bel58] Richard Ernest Bellman. On a Routing Problem. *Quarterly of applied mathematics*, 16:87–90, 1958.
- [BGP99] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.
- [Bla05] Paul E. Black. Gray code, in dictionary of algorithms and data structures. See <http://www.nist.gov/dads/HTML/graycode.html>, 2005.
- [BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.
- [BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN*, pages 103–122, Toronto, Canada, 2001. Springer.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, Los Angeles, USA, 1977. ACM.
- [CFR⁺89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of POPL (ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages)*, pages 25–35. ACM, 1989.
- [CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–4, Edinburgh, UK, 2005. Springer.
- [Cla00] Edmund Clarke. *Model Checking*. MIT Press, Boston, USA, January 2000.
- [CR06] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196, Vienna, Austria, 2006. Springer.
- [DNS03] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a Theorem Prover for Program Checking. Technical Report 148, HP Labs, 2003.
- [Fla04] Cormac Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. *CP+CV'04*, 2004.

- [FLL⁺02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
- [Gat02] Bill Gates. Keynote address at WinHEC'02, 2002. Available at <http://www.microsoft.com/presspass/exec/billg/speeches/2002/04-18winhec.aspx>.
- [HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, Portland, USA, 2002. ACM.
- [ISGG05] Franco Ivanicic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking c programs using f-soft. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.
- [KCY03] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proc. of DAC 2003*, pages 368–371, Anaheim, USA, 2003. ACM Press.
- [Knu97] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3. Addison-Wesley, 1997.
- [Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *JSAT*, 3(3-4):141–224, 2007.