

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

Alessandro Armando   Jacopo Mantovani   Lorenzo Platania

Artificial Intelligence Laboratory  
DIST, University of Genova, Italy

31st March 2006



# Bounded Model Checking

Bounded Model Checking is a successful approach to automatic analysis of systems. Its key ideas are:

- **building a quantifier-free formula** whose models correspond to system traces that violate some given property, and
- **deciding the satisfiability** of the formula.

In software analysis, the state-of-the-art tool in this area is **CBMC**, which:

- builds a **boolean formula**, and
- decides its satisfiability using a state-of-the-art **SAT solver**.

# Bounded Model Checking

Bounded Model Checking is a successful approach to automatic analysis of systems. Its key ideas are:

- ➊ **building a quantifier-free formula** whose models correspond to system traces that violate some given property, and
- ➋ **deciding the satisfiability** of the formula.

In software analysis, the state-of-the-art tool in this area is **CBMC**, which:

- ➊ builds a **boolean formula**, and
- ➋ decides its satisfiability using a state-of-the-art **SAT solver**.

# Bounded Model Checking of Software

Instead of encoding the verification problem as a SAT problem, we propose the use of combination of richer (but still decidable) theories to **reduce the verification of software to a Satisfiability Modulo Theories (SMT) problem** i.e.:

$$\text{SAT}(\mathcal{T} \cup \{\Phi\})$$

where

- $\mathcal{T}$  is a decidable theory, and
  - $\Phi$  is a quantifier-free formula in the same language as  $\mathcal{T}$ .
- Our encoding leads to a formula in the combination of the theories of **bitvectors**, **records**, and **arrays** for which efficient SMT solvers exist.
  - We show that the usage of SMT solvers instead of SAT solvers may lead to:
    - a considerably **smaller encoding**,
    - **better performance** on problems of interests.

# Bounded Model Checking of Software

Instead of encoding the verification problem as a SAT problem, we propose the use of combination of richer (but still decidable) theories to **reduce the verification of software to a Satisfiability Modulo Theories (SMT) problem** i.e.:

$$\text{SAT}(\mathcal{T} \cup \{\Phi\})$$

where

- $\mathcal{T}$  is a decidable theory, and
  - $\Phi$  is a quantifier-free formula in the same language as  $\mathcal{T}$ .
- Our encoding leads to a formula in the combination of the theories of **bitvectors**, **records**, and **arrays** for which efficient SMT solvers exist.
  - We show that the usage of SMT solvers instead of SAT solvers may lead to:
    - a considerably smaller encoding,
    - better performance on problems of interests.

# Bounded Model Checking of Software

Instead of encoding the verification problem as a SAT problem, we propose the use of combination of richer (but still decidable) theories to **reduce the verification of software to a Satisfiability Modulo Theories (SMT) problem** i.e.:

$$\text{SAT}(\mathcal{T} \cup \{\Phi\})$$

where

- $\mathcal{T}$  is a decidable theory, and
  - $\Phi$  is a quantifier-free formula in the same language as  $\mathcal{T}$ .
- Our encoding leads to a formula in the combination of the theories of **bitvectors**, **records**, and **arrays** for which efficient SMT solvers exist.
  - We show that the usage of SMT solvers instead of SAT solvers may lead to:
    - a considerably **smaller encoding**,
    - **better performance** on problems of interests.

# A different encoding: the theory of arrays

Arrays are nothing but functions that bind indexes to elements.

Function symbols *select* and *store* model access to and storage of elements. They are defined as follows:

## Axioms

$$\forall a : Array, \forall i, j : Index, \forall e : Element,$$

$$select(store(a, i, e), i) = e,$$

$$i \neq j \supset select(store(a, i, e), j) = select(a, j).$$

# A different encoding: the theory of records

Records are functions that bind field identifiers  $Id_k$  to elements of type  $T_k$ :

$$Id : \{Id_1, \dots, Id_n\}$$

$$r : Id_k \rightarrow T_k, \forall k = 1, \dots, n.$$

Function symbols  $rselect_k$  e  $rstore_k$ , for  $k = 1, \dots, n$  model access to and storage of record fields. They are defined as follows:

## Axioms

$$\forall r : Record, \forall Id_k \in Id,$$

$$rselect_k(rstore_k(r, e)) = e \quad \text{for } k = 1, \dots, n.$$

$$rselect_l(rstore_k(r, e)) = rselect_l(r) \quad \text{for } 1 \leq k \neq l \leq n.$$

$$\text{where } e \in T_k$$

# A different encoding: the theory of bit-vectors

Bit-vectors associate boolean values to a finite set of indexes. The theory defines a rich family of function symbols consisting of:

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

# The Procedure

Let  $P$  be the input program, and let  $k$  be the given bound. Our Bounded Model Checking procedure, inspired by CBMC, consists of

- preprocessing  $P$ :
  - removing all control statements except `while` and `if`.
  - unwinding the `while` loops  $k$  times,
  - removing all side effects and putting the program in Single Assignment Form,
- generating an SMT formula encoding the program and the desired properties,
- deciding the satisfiability of the formula.

# The Procedure

Let  $P$  be the input program, and let  $k$  be the given bound. Our Bounded Model Checking procedure, inspired by CBMC, consists of

- 1 preprocessing  $P$ :
  - removing all control statements except `while` and `if`.
  - unwinding the `while` loops  $k$  times,
  - removing all side effects and putting the program in Single Assignment Form,
- 2 generating an SMT formula encoding the program and the desired properties,
- 3 deciding the satisfiability of the formula.

Let  $P$  be the input program, and let  $k$  be the given bound. Our Bounded Model Checking procedure, inspired by CBMC, consists of

- 1 preprocessing  $P$ :
  - removing all control statements except `while` and `if`.
  - unwinding the `while` loops  $k$  times,
  - removing all side effects and putting the program in Single Assignment Form,
- 2 generating an SMT formula encoding the program and the desired properties,
- 3 deciding the satisfiability of the formula.

# The Procedure – Preprocessing (1)

The program is transformed into an equivalent one that uses only `while` and `if`.

```
...
for(i=0;i<2;i++)
{
  a[i]=i;
}
...

...
i=0;
while(i<2)
{
  a[i]=i;
  i++;
}
...
```

## The Procedure – Preprocessing (2)

**Unwinding the loops** ( $k$  times): every loop is reduced to a sequence of  $k$  nested `if` statements:

```
...
while(i<2)
{
  a[i]=i;
  i++;
}
...

...
if(i<2)
{
  a[i]=i;
  i++;
  if(i<2)
  {
    a[i]=i;
    i++;
    assert(!i<2);
  }
}
...
```

$\xrightarrow{k=2}$

## The Procedure – Preprocessing (3)

**Renaming program variables:** all side effects are removed and every program variable is renamed to put the program in Single Assignment Form:

```
...
i=0;
if(i<2)
{
  a[i]=i;
  i++;
  if(i<2)
  {
    a[i]=i;
    i++;
    assert(!i<2);
  }
}
...
```

$\xrightarrow{\rho}$

```
...
i1=0;
if(i1<5)
{
  a1[i1]=i1;
  i2=i1+1;
  if(i2<5)
  {
    a2[i2]=i2;
    i3=i2+1;
    assert(!i3<5);
  }
}
...
```

# The procedure – Generating the formula

The generation of the formula  $\Phi$ : we encode the the preprocessed program into a quantifier-free formula  $\Phi = \mathcal{C} \wedge \neg \mathcal{P}$  whose models correspond to program traces that violate some of the given properties;

$$\begin{aligned}\mathcal{C} &= (true \supset i_1 = 0) \wedge (false \supset i_1 = i_0) \wedge \\ & ((i_1 < 2) \supset a_1 = store(a_0, i_1, i_1)) \wedge (\neg(i_1 < 2) \supset a_1 = a_0) \wedge \\ & ((i_1 < 2) \supset i_2 = i_1 + 1) \wedge \neg(i_1 < 2) \supset i_2 = i_1) \wedge \\ & ((i_1 < 2) \wedge (i_2 < 2) \supset a_2 = store(a_1, i_2, i_2)) \wedge \\ & \neg((i_1 < 2) \wedge (i_2 < 2)) \supset a_2 = a_1) \wedge \\ & ((i_1 < 2) \wedge (i_2 < 2) \supset i_3 = i_2 + 1) \wedge \\ & \neg((i_1 < 2) \wedge (i_2 < 2)) \supset i_3 = i_2) \\ \mathcal{P} &= ((i_1 < 2) \wedge (i_2 < 2)) \supset \neg(i_3 < 2)\end{aligned}$$

## SAT encoding

CBMC encodes  $\Phi$  as a boolean formula where:

- variables of basic type are modelled as vectors of  $n$  bits,
- arrays of  $m$  elements are modelled as  $m$  different variables (vectors of  $n$  bits, i.e.  $n \times m$  bits),

and uses the Chaff solver to decide its satisfiability.

## SMT encoding

Our approach encodes  $\Phi$  as an SMT formula where

- variables of basic type are modelled as bit-vector variables,
- arrays of  $m$  elements are modelled as... arrays!,

and uses CVC Lite to decide its satisfiability.

## SAT encoding

CBMC encodes  $\Phi$  as a boolean formula where:

- variables of basic type are modelled as vectors of  $n$  bits,
- arrays of  $m$  elements are modelled as  $m$  different variables (vectors of  $n$  bits, i.e.  $n \times m$  bits),

and uses the Chaff solver to decide its satisfiability.

## SMT encoding

Our approach encodes  $\Phi$  as an SMT formula where

- variables of basic type are modelled as bit-vector variables,
- arrays of  $m$  elements are modelled as... arrays!,

and uses CVC Lite to decide its satisfiability.

# SMT vs. SAT (1)

Using the functional symbols from the Theory of Arrays results in a more compact encoding for all array accesses:

Statement	SMT encoding	SAT encoding
$a[e_1] = e_2$	$a_k = \text{store}(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} a_k^i = ((i = e_1) ? e_2 : a_{k-1}^i)$
$x = a[e]$	$x_j = \text{select}(a_k, e)$	$(\bigwedge_{i=0}^{\text{dim}(a)-1} (i = e) \supset v = a_k^i) \wedge (x_j = v)$

- SAT encoding grows with the size of the arrays;
- SMT encoding is independent from the size of the arrays.

# SMT vs. SAT (1)

Using the functional symbols from the Theory of Arrays results in a more compact encoding for all array accesses:

Statement	SMT encoding	SAT encoding
$a[e_1] = e_2$	$a_k = \text{store}(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{\dim(a)-1} a_k^i = ((i = e_1) ? e_2 : a_{k-1}^i)$
$x = a[e]$	$x_j = \text{select}(a_k, e)$	$(\bigwedge_{i=0}^{\dim(a)-1} (i = e) \supset v = a_k^i) \wedge (x_j = v)$

- SAT encoding grows with the size of the arrays;
- **SMT encoding is independent from the size of the arrays.**



# Experimental Results (1)

- In order to assess the effectiveness of our approach we have developed a prototype implementation called **SMT-CBMC**.
- We have run **SMT-CBMC** against a number of families of C programs.
- Each family is parametric in a positive integer  $N$  such that both the size of the arrays and the bound depend on  $N$ .
- We have run both **SMT-CBMC** and **CBMC** on a number of benchmark programs:
  - *Bubble Sort*( $N$ ),
  - *Selection Sort*( $N$ ),
  - *Bellman Ford*( $N$ ), and
  - *Prim*( $N$ ).
- The programs involve a tight interplay between arithmetics and array manipulation, which is a common characteristic in programming.

# Experimental Results (2)

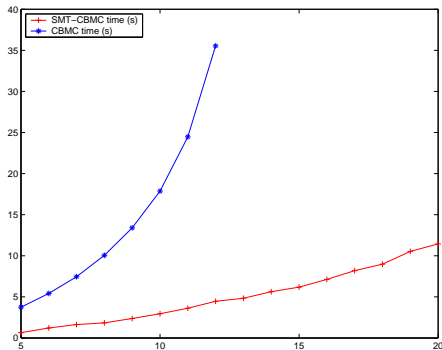
## BellmanFord( $N$ )

```
.....
for(i = 0; i < edgecount; i++){
  assume(Weight[i]=i);
  assume(Dest[i]>=0);
  assume(Source[i]>=0);
  assume(Dest[i]<nodecount);
  assume(Source[i]<nodecount);
}
for(i = 0; i < nodecount; i++){
  if(i == source)
    distance[i] = 0;
  else
    distance[i] = INFINITY;
}
```

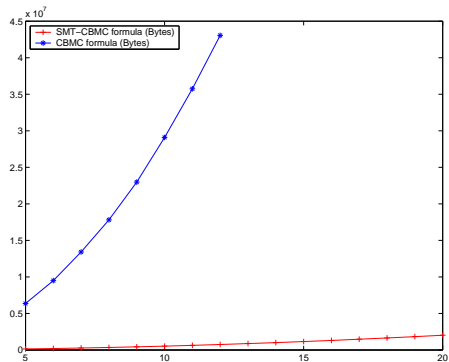
```
for(i = 0; i < nodecount; i++){
  for(j = 0; j < edgecount; j++){
    x = Dest[j];
    y = Source[j];
    if(distance[x] > distance[y]+Weight[j])
      distance[x] = distance[y] + Weight[j];
  }
}
for(i = 0; i < edgecount; i++){
  x = Dest[i];
  y = Source[i];
  if(distance[x] > distance[y]+Weight[i])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

# Experimental Results (3)

BellmanFord( $N$ )



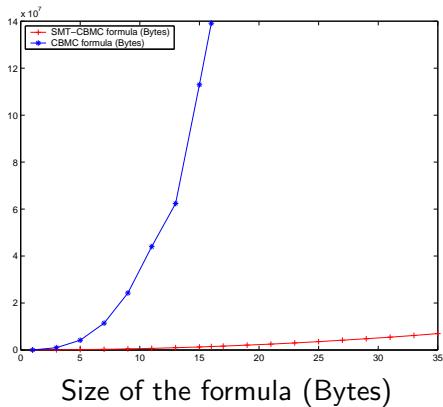
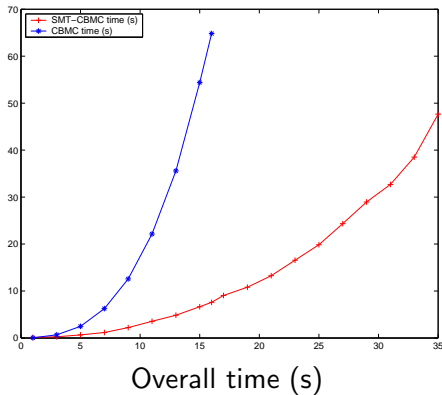
Overall time (s)



Size of the formula (Bytes)

# Experimental Results (4)

Bubblesort( $N$ )



# Experimental Results (5)

Prim( $N$ )

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

- We have proposed the use of SMT solvers in software Bounded Model Checking instead of SAT solvers.
- This results in **more compact formulae** when arrays are involved since the size of our formula does not depend on the size of the involved arrays.
- The experimental results confirm the effectiveness of our approach: **SMT-CBMC performs better** than CBMC on programs using arrays.

`www.ai.dist.unige.it/eureka`