

# Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver

A. Armando   J. Mantovani   L. Platania

Artificial Intelligence Laboratory  
DIST - University of Genova

Cambridge – September 12, 2005



- Bounded Model Checking (basic idea):
  - ① reduce a bounded model checking problem to a satisfiability problem in propositional logic (SAT), then
  - ② use a state-of-the-art SAT solver to solve the problem.
- Initially applied successfully to analyze HW circuits
- Recently applied to find bugs in sequential programs (CBMC)
- However model checking of SW poses new challenges as programs often deal with large or potentially unbounded data.
- We have adapted the CBCM approach to SW Model Checking to use a SMT solver instead of a SAT solver.
- We show that the usage of SMT solvers instead of SAT solvers may improve performance considerably on problems of interests.



- Bounded Model Checking (basic idea):
  - ① reduce a bounded model checking problem to a satisfiability problem in propositional logic (SAT), then
  - ② use a state-of-the-art SAT solver to solve the problem.
- Initially applied successfully to analyze HW circuits
- Recently applied to find bugs in sequential programs (CBMC)
- However model checking of SW poses new challenges as programs often deal with large or potentially unbounded data.
- We have adapted the CBCM approach to SW Model Checking to use a SMT solver instead of a SAT solver.
- We show that the usage of SMT solvers instead of SAT solvers may improve performance considerably on problems of interests.



# Bounded Model Checking of Sequential Software

The procedure used in CMBC consists of 4 steps:

- 1 **Preprocess**: the program is transformed into an equivalent one that uses only `while` and `if`.
- 2 **Unwind loops** ( $n$  times):

$$\begin{aligned} \text{while}(e) \text{ instr} &\longrightarrow \text{if}(e) \{ \text{instr}; \text{while}(e) \text{ instr} \} \\ \text{while}(e) \text{ instr} &\longrightarrow \text{assert}(!e); \end{aligned}$$

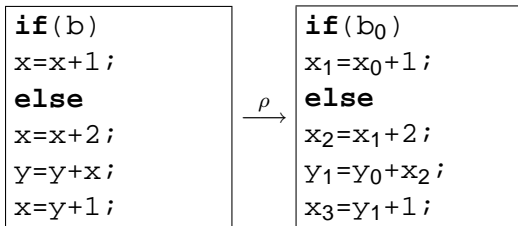
- 3 **Build** a quantifier-free formula  $\Phi$  such that any model of  $\Phi$  corresponds to an execution path ending with the violation of an `assert` statement.
- 4 **Check** whether  $\Phi$  is satisfiable. If any unwinding assertion is violated, then increase  $n$  and go to 2 otherwise return result of the analysis.



# Renaming

As a preliminary step to the construction of  $\Phi$ , program variables are systematically renamed so that each renamed variable is assigned only once.

For example:



Let  $p$  the renamed program and  $g$  a formula representing the initial states.

Then  $\Phi := (\mathcal{C}(p, g) \wedge \neg \mathcal{P}(p, g))$ , where

- $\mathcal{C}(p, g)$  represents the set of execution paths of  $p$  starting in an initial state and
- $\mathcal{P}(p, g)$  represents the set of states for which the `assert` statements succeed.

Thus, any model of  $\Phi$  corresponds to an execution path that starts in an initial state and violates an `assert` statement.



# Computing $\mathcal{C}(p, g)$ and $\mathcal{P}(p, g)$

The formula  $\mathcal{C}(p, g)$  is recursively defined as follows:

- If  $p$  is empty, then  $\mathcal{C}(p, g) := \text{true}$
- $\mathcal{C}(\mathbf{if}(c) \ i \ \mathbf{else} \ i', g) := \mathcal{C}(i, g \wedge c) \wedge \mathcal{C}(i', g \wedge \neg c)$
- $\mathcal{C}(i; i', g) := \mathcal{C}(i, g) \wedge \mathcal{C}(i', g)$
- $\mathcal{C}(v=e, g) := (v_k = (g ? e : v_{k-1}))$
- $\mathcal{C}(a[e]=e', g) := (a_k = (g ? \text{store}(a_{k-1}, e, e') : a_{k-1}))$

$\mathcal{P}(p, g)$  is defined in a similar way.



# Computing $\mathcal{C}(p, g)$ and $\mathcal{P}(p, g)$

The formula  $\mathcal{C}(p, g)$  is recursively defined as follows:

- If  $p$  is empty, then  $\mathcal{C}(p, g) := \text{true}$
- $\mathcal{C}(\mathbf{if}(c) \ i \ \mathbf{else} \ i', g) := \mathcal{C}(i, g \wedge c) \wedge \mathcal{C}(i', g \wedge \neg c)$
- $\mathcal{C}(i; i', g) := \mathcal{C}(i, g) \wedge \mathcal{C}(i', g)$
- $\mathcal{C}(v=e, g) := (v_k = (g ? e : v_{k-1}))$
- $\mathcal{C}(a[e]=e', g) := (a_k = (g ? \text{store}(a_{k-1}, e, e') : a_{k-1}))$

$\mathcal{P}(p, g)$  is defined in a similar way.



# Solving $\Phi$ with a SAT solver

- Basic data types (e.g. `int`, `float`) modeled as fixed size bit-vectors.  
Arithmetical operations are thus converted into equivalent combinations of bit-vectors operations.
- Arrays with  $m$  elements modeled as  $m$  variables.  
Equations involving *store* operations are rewritten as follows:

$$\begin{aligned} a_k &= \text{store}(a_{k-1}, e, e') \\ &\Downarrow \\ \bigwedge_{i=0}^{\text{dim}(a)-1} a_k[i] &= ((i = e) ? e' : a_{k-1}[i]) \end{aligned}$$

- The resulting formula is a boolean combination of bit-vector equations that can be readily compiled into an “equivalent” propositional formula.



# Solving $\Phi$ with a SAT solver – continued

## Pros:

- State-of-the-art SAT solvers can be used to solve the formula efficiently.
- Faithful modeling of the data and operations upon them.

## Cons:

- Result of the analysis depends on the chosen size of bit-vectors used to model basic data types.
- Size of the encoding increases with the size of the arrays used in the program.

Note: This is essentially the approach adopted by CBMC.



# Solving $\Phi$ with a SMT solver

Two alternatives are possible:

(A1) As before, but model array as ... **arrays**!

- The resulting formula belongs to the combination of the theory of bit-vectors and the theory of arrays and can be checked by existing SMT solvers (e.g. CVC-Lite).

(A2) As (A1) + basic data types (e.g. **int**, **float**) modeled by corresponding numerical domains ( $\mathbb{Z}$ ,  $\mathbb{R}$ , resp.).

- In this case (if the program does not contain any multiplication between variables) then the resulting formula belongs to the combination of linear arithmetics and the theory of arrays and can be effectively checked by existing SMT solvers.



# Solving $\Phi$ with a SMT solver – continued

## Pros:

- State-of-the-art SMT solvers can be used to solve the formula efficiently.
- Size of the encoding independent from the size of the arrays used in the program.
- (A2) Result of the analysis independent from the actual binary representation of basic data types.

## Cons:

- (A2)  $\mathbb{Z}$  and  $\mathbb{R}$  do not model `int` and `float` in a faithful way.
  - No support for modular arithmetics.
  - However, overflows can be easily detected by adding assertions to the original program.
- (A2) Only programs involving linear arithmetic expressions can be analysed efficiently.

## Question: Performance?

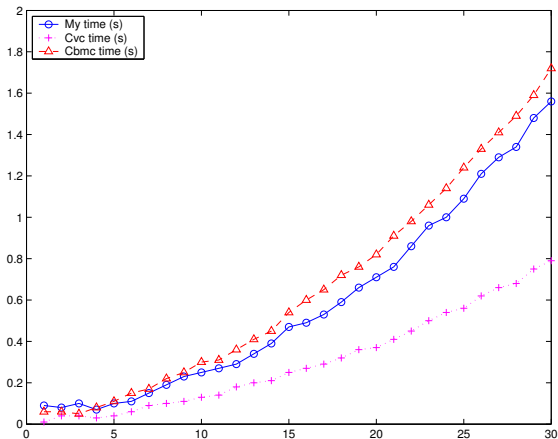


- We have developed a prototype implementation of (A2).
- Formulae generated are fed to CVC-Lite.
- To assess scalability we have considered programs for increasing values of a parameter **N**.



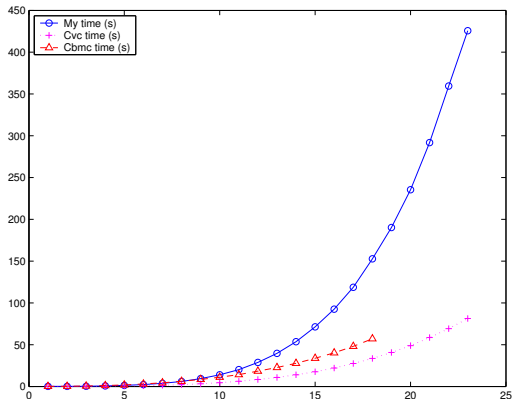
# Experiments: swap\_iter(N)

```
int x,y;
void main() {
  int k;
  k=0;
  x=10;
  y=20;
  while(k<N) {
    swap();
    k=k+1; }
  if(x==0)
  assert(0); }
void swap() {
  int c;
  c=x;
  x=y;
  y=c; }
```



# Experiments: InsertSort (N)

```
int a[N];  
  
int InsertSort() {  
  int i, j, t;  
  for (j=1;j<7;j++) {  
    t=a[j];  
    for(i=j-1;i>=0&& t<a[i];i--)  
      a[i+1]= a[i];  
    a[i+1] = t; } }  
  
int main() {  
  int i;  
  for(i=0;i<N;i++)  
    a[i]=N-1-i;  
  InsertSort();  
  for(i=0;i<N;i++)  
    assert(a[i]==i);
```

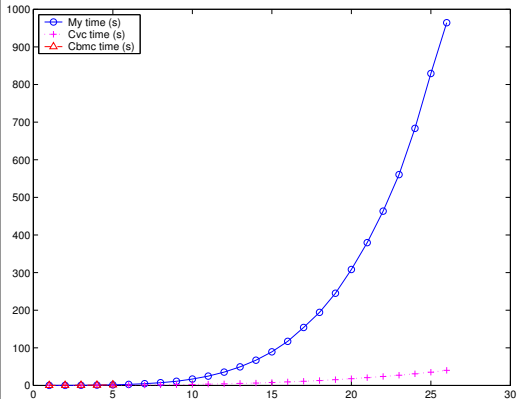


# Experiments: SelectSort(N)

```
int a[N];

int 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; }

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



# Conclusions & Ongoing Work

- We have adapted the CBCM approach to SW Model Checking to use a SMT solver instead of a SAT solver.
- Our preliminary results confirm that SMT solvers can compete and in some cases significantly outperform SAT solvers in Bounded Model Checking of SW.
- Development of (A2) is under way.

