Bounded Model Checking

Michel Rueher

Bounded Model Checking framework

- **Models** → finite automates, labelled transition systems
- **Properties**:
  - Safety → something bad should not happen
  - Liveness → something good should happen
- **Bound *k*** → look only for counter examples made of *k* states

Bounded Model Checking framework (cont.)

% set of states: S, initial states: I, transition relation: T

*% **bad states B reachable from I via T?***

bounded_model_checker$_{forward}$(I,T,B,k)

SC =ø; SN =I; n=1

while $S_C \neq S_N$ **and** n<k **do**

If B ∩ $S_N \neq \emptyset$ **then return** "*found error trace to bad states*";

else $S_C = S_N$; $S_N = S_C \cup T(SC)$; $n = n + 1$;

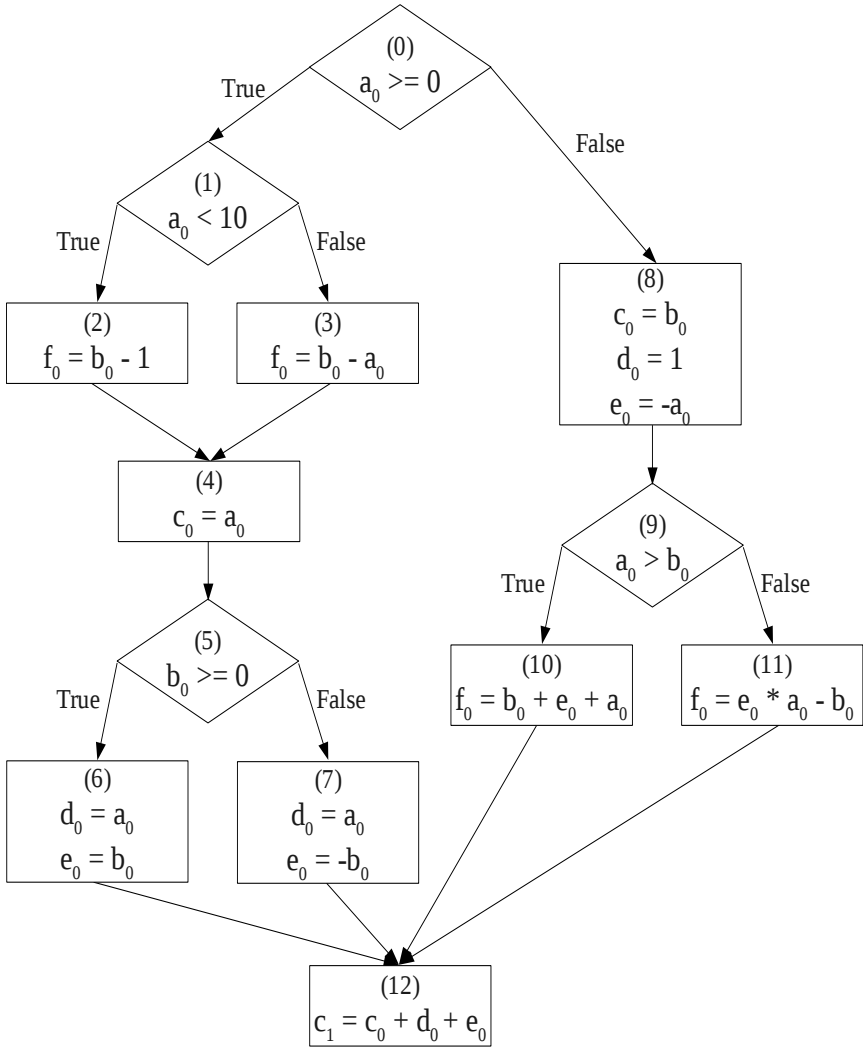
done

return "*no bad state reachable*";

Bounded Model Checking: slicing (example)

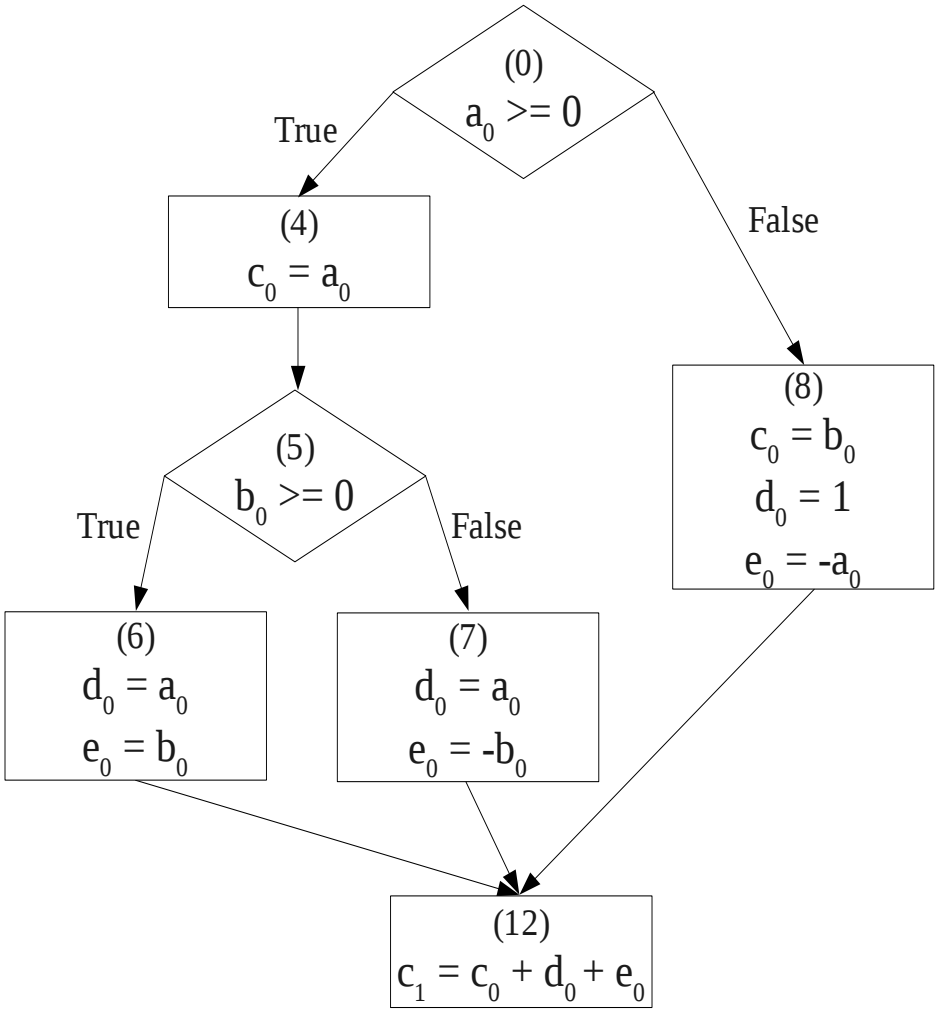
```
void foo(int a, int b)
int c, d, e, f;
if (a >= 0) {
   if (a<10)  {f =b-1;}  else  {f =b-a;}
   c = a;
   if (b>=0)  {d =a;e=b;} else{d =a;e=-b;} }
else  {c = b; d = 1; e = -a;
   if(a>b) {f =b+e+a;} else{f =e*a-b;} }
c = c + d + e;
assert(c >= d + e); // property p1
assert(f >= -b * e); // property p2
```

BMC: slicing (cont.)




```
void foo(int a, int b)
int c, d, e, f;
if(a >= 0) {
      if(a < 10) {f = b – 1;}
      else {f = b – a; }
      c = a;
      if(b >= 0) {d = a; e = b;}
      else {d = a; e = –b;}   }
  else {
      c = b; d = 1; e = –a;
      if(a > b) {f = b + e + a;}
      else {f = e * a – b;}   }
  c = c + d + e;
  assert(c >= d + e); // property p1
  assert(f >= –b * e); // property p2
```

BMC: slicing /example (cont.)




```
void foo(int a, int b)
int c, d, e, f;
if(a >= 0) {
      if(a < 10) {f = b − 1;}
      else {f = b − a;}
      c = a;
      if(b >= 0) {d = a; e = b;}
      else {d = a; e = −b;}   }
  else {
      c = b; d = 1; e = −a;
      if(a > b) {f = b + e + a;}
      else {f = e ∗ a − b;}   }
  c = c + d + e;
  assert(c >= d + e); // property p1
  assert(f >= −b ∗ e); // property p2
```

SAT/SMT- based BMC: Bounded Model Checking

1. The **program is unwound *k*** times

2. The unwound (and simplified) program and the property are translated into

   A big propositional formula φ

   [arithmetic formula → bit vector encoding]

   % φ is satisfiable iff there exists a counterexample of depth less than k

3. **A SAT or SMTsolver** is used for checking the satisfiability of φ

CP-based Bounded Program Verification

1. The **program is unwound *k*** times,

2. An **annotated** and **simplified CFG** is built

3. Program is translated in constraints **on the fly**

   → A **list of solvers** tried in sequence (LP, MILP, Boolean, CP)

Constraint Generation

```
Input : i0
    j0 = 2
    if( i0 ≤ 16 )
          j1 = j0 * i0
      j2 = j1
    else
      j2 = j0

    if( j2 > 8)
      j3 = 0
    else
      j3 = j2
    r =  j3
Output : r
```

Variable : i_0

$j_0 = 2$

$i_0 \leq 16 \Rightarrow (j_1 = j_0 * i_0 \wedge j_2 = j_1)$

$i_0 > 16 \Rightarrow j_2 = j_0$

$\neg (j_1 = j_0 * i_0 \wedge j_2 = j_1) \Rightarrow (i_0 > 16 \wedge j_2 = j_0)$

$\neg (j_2 = j_0) \Rightarrow (i_0 \leq 16 \wedge j_1 = j_0 * i_0 \wedge j_2 = j_1)$

$j_2 > 8 \Rightarrow j_3 = 0$

$j_2 \leq 8 \Rightarrow j_3 = j_2$

$\neg (j_3 = 0) \Rightarrow (j_2 \leq 8 \wedge j_3 = j_2)$

$\neg (j_3 = j_2) \Rightarrow (j_2 > 8 \wedge j_3 = 0)$

$r = j_3$

Variable: r

Constraint Generation (cont.)

Program:

```
x=x+1; y=x*y; x=x+y;
```

Constraints

```
{x1 = x0 +1,y1 =x1 *y0,x2 =x1 *y1}
```

Constraint Generation (cont.)

Program:

```
a[i] = x;
```

Constraints

```
{a1[i0] = x0,
   i0 ≠ 0 → a1[0] = a0[0],
   i0 ≠ 1 → a1[1] = a0[1],...,
   i0 ≠ 7 → a1[7] = a0[7]}
```

guard → *body* is a **guarded constraint**

a[i] = *x* is the **element constraint:** i and x are constrained variables whose values may be unknown

Constraint Generation (cont.)

Program:

```
if (a<10) {f =b-1;}    else {f =b-a;}
```

Constraints

$C \wedge (a<10) \rightarrow \{f0 = b0-1\}$

$C \wedge \neg(a<10) \rightarrow \{f0 = b0-a0\}$

Error Localization problem: informal presentation

- **Model checking**, testing

  Generation of counterexamples

  - Input data & wrong output (testing)
  - Input data & violated post condition / property
    → Execution trace

- **Problems**:

  - Execution trace: often *lengthy* and *difficult to understand*
  - Location of the portions of code that contain errors
    → *Very expensive*

Constraint-Based Error Localization: Formalization

- **P**: program
- **Post_P**: post condition of P
- **Pre_P**: precondition of P
- **CST_P**: constraints of faulty path of P (Input data provided by Model checker)
  → **Pred_P ∧ CST_P ∧ ¬ Post_P** holds
  → **Pred_P ∧ CST_P ∧ Post_P** fails

Problem: to finding "smallest" subsets of **Pred_P ∧ CST_P ∧ Post_P** that are inconsistent

Example

Program:

```
% Input : int input1, int input2
int x = 1, y = 1, z = 1;
if (input1 > 0) {x += 5; y += 6; z += 1; }
if (input2 > 0) {x += 6; y += 5; z += 4; }
% Post-condition: x < 10 ∧ y < 10
```

Counterexample: *input1=1, input2=1*

CSP P:

input1=1, input2=1, $x_{10} = 1$, $y_{10} = 1$,

$z_{10} = 1$, $x_{11} = 6$, $y_{11} = 7$, $z_{11} = 2$, $x_{12} = x_{11}$, $y_{12} = y_{11}$, $z_{12} = z_{11}$,

$x_{13} = x_{12}+6$, $y_{13} = y_{12}+5$, $z_{13} = z_{12}+4$, $x_{14} = x_{13}$, $y_{14} = y_{13}$, $z_{14} = z_{13}$,

$x_{14} < 10$, $y_{14} < 10$

Example (cont.)

CS_P can be divided into 3 sub-CSPs (computations for x, y, and z are independent)

sub CSP_x is: $x_{10} = 1$, $x_{11} = 6$, $x_{12} = x_{11}$, $x_{13} = x_{12}+6$, $x_{14} = x_{13}$, $x_{14} < 10$

sub CSP_y is: $y_{10} = 1$, $y_{11} = 7$, $y_{12} = y_{11}$, $y_{13} = y_{12}+5$, $y_{14} = y_{13}$, $y_{14} < 10$

Smallest inconsistent CSP for x: ~~$x_{10} = 1$~~, $x_{11} = 6$, $x_{12} = x_{11}$, $x_{13} = x_{12}+6$, $x_{14} = x_{13}$, $x_{14} < 10$

Smallest inconsistent CSP, for y: ~~$y_{10} = 1$~~, $y_{11} = 7$, $y_{12} = y_{11}$, $y_{13} = y_{12}+5$, $y_{14} = y_{13}$, $y_{14} < 10$

A first solution: MAX-SAT

MAX-SAT based appoach
(implemented in Bug-Assist with CBMC)

1. Encoding a trace of a program as a Boolean formula F that is satisfiable iff the trace is satisfiable

2. Building a failing formula F' by asserting that the post condition must hold

3. Computing with MAX-SAT the maximum number of clauses that can be satisfied in F'
   → *complement as a potential cause of the errors*

Generalisation of MAX-SAT

- **Capabilities** of CP, LP, MIP: No Boolean abstraction (or bit vector encoding) required to capture the semantics of the constraints

  → ***Generalisation of MAX-SAT approach***

  - **IIS**
  - **Minimum Conflict Sets in CSP**

Definitions

- **MUS Minimal Unsatisfiable Subset**
  ***aka* Irreducible Inconsistent Subsystem (IIS)**
  $M \subseteq C$ **is a MUS** $\Leftrightarrow$ M **is UNSAT and** $\forall c \in M : M \setminus \{c\}$ **is SAT**

- **MSS Maximal Satisfiable Subset**
  a generalization of MaxSAT / MaxFS considering maximality instead of maximum cardinality
  $M \subseteq C$ **is a MSS** $\Leftrightarrow$ M **is SAT and** $\forall c \in C \setminus M : M \cup \{c\}$ **is UNSAT**

- **MCS Minimal Correction Set**
  the complement of some MSS: removal yields a satisfiable MSS (it "corrects" the infeasibility)
  $M \subseteq C$ **is a MCS** $\Leftrightarrow$ $C \setminus M$ **is SAT and** $\forall c \in M : (C \setminus M) \cup \{c\}$ **is UNSAT**

MUS (Minimal Unsatisfiable Subset) **- MCS**(Minimal Correction Set) **duality**

The set of MCSes ⇔ all the irreducible hitting sets of the MUSes

The set of MUSes ⇔ The set of all irreducible hitting sets of the MCSes

H is a hitting set of Ω if $H \subseteq D$ and $\forall S \in \Omega, H \cap S \neq \varnothing$

H is a minimal hitting sets if no element can be removed without losing the the property of being a hitting set

Given an unsatisfiable constraint system C:

1. A subset M of C is an MCS of C iff M is an irreducible hitting set of MUSes(C)
2. A subset U of C is an MUS of C iff U is an irreducible hitting set of MCSes(C)

Intuition: A MCS *must at least remove one constraint* from each MUS

A MUS can be made satisfiable by removing any one constraint from it

→ every MCS contains at least one constraint from each MUS.

IIS/MUS – Problems and challenges

Problems:

- The number of IISs in an infeasible LP can be ***exponential*** in the worst case
- Quickest algorithms for finding IISs often return IISs having many rows

IIS – Algorithms

The Deletion Filter:

INPUT: an infeasible set of constraints

FOR each constraint in the set:

1. Temporarily drop the constraint from the set
2. Test the feasibility of the reduced set:

IF feasible THEN return dropped constraint to the set

ELSE (infeasible) drop the constraint permanently

OUTPUT: constraints constituting **a single IIS**

Remarks

- The only constraints retained in the set are those whose removal renders the set feasible
- Efficiency improvement: dynamic reordering

The Elastic Filter – Linear constraints (1)

LP solvers

- Adding *nonnegative artificial variables* (s_i) to all inequality constraints
- LP Phase 1 minimizes $W = \sum s_i$, via standard LP: If $W^* \neq 0$, no solution exists

Elastic Filter: *nonnegative artificial variables* (s_i) are added to all equality and ≥ constraints

→ so a solution always exits in the space of the original plus artificial variables, but not in the space of just the original variables

If $W^* \neq 0$ then at least one of s_i cannot be forced to zero: the corresponding constraint remains violated in the original variable space

Rules for adding elastic variables are as follows:

non-elastic constraint	elastic version
$\Sigma_j a_{ij} x_j \geq b_i$	$\Sigma_j a_{ij} x_j + s_i \geq b_i$
$\Sigma_j a_{ij} x_j \leq b_i$	$\Sigma_j a_{ij} x_j - s_i \leq b_i$
$\Sigma_j a_{ij} x_j = b_i$	$\Sigma_j a_{ij} x_j + s_i - s_i = b_i$

IIS – Algorithms (cont.)

Use the concept of "elastic programming": ***non-negative "elastic variables"*** are added to the constraints to provide elasticity

Non-elastic constraint	Elastic constraint
$\Sigma_j\, a_{ij}x_i \geq b_t$	$\Sigma_j\, a_{ij}x_i \mathbf{+ e_t} \geq b_t$
$\Sigma_j\, a_{ij}x_i \leq b_t$	$\Sigma_j\, a_{ij}x_i \mathbf{- e_t} \leq b_t$
$\Sigma_j\, a_{ij}x_i = b_t$	$\Sigma_j\, a_{ij}x_i \mathbf{+ e'_t - e''_t} = b_t$

IIS – Algorithms (cont.)

The Elastic Filter:

% Input : an infeasible set of linear constraints

1. Make all constraints elastic by adding non-negative elastic variables
2. Solve LP using elastic objective function

IF feasible THEN enforce the constraints with any $e_t > 0$ by

permanently removing their elastic variable(s)

GO TO step 2

ELSE *(% infeasible)* EXIt

END FOR

OUTPUT : the set of enforced constraints contains at least one IIS

Computing all MCS : CAMUS

All_MCSes(φ)

1. **$\varphi' \leftarrow$ AddYVars(φ)** *% Adds y_i selector variables*
2. **MCSes ← ∅**
3. **k ← 1**
4. **while (SAT(φ'))**
5. **$\varphi'_k \leftarrow \varphi' \circ$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$, k)**
6. **while (newMCS ← IncrementalSAT(φ'_k))** *%All MCS of size K*
7. **MCSes ← MCSes ∪ {newMCS}**
8. **$\varphi'_k \leftarrow \varphi'_k \circ$ BlockingClause(newMCS)** % Excludes super sets for *% for size k*
9. **$\varphi' \leftarrow \varphi' \circ$ BlockingClause(newMCS)** *% Excludes super set % for size > k*
10. **end while**
11. **k←k+1**
12. **end while**
13. **return MCSes**

Computing all MCS – Example

- $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$
- $\phi = (x_1) \wedge (\neg x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$
- $\phi' = (\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3$
- **K = 1**

  $\neg y_1 \wedge \neg x_1 \wedge (\neg x_1 \vee x_2) \vee \neg x_2 \vee (\neg x_1 \vee x_3) \wedge \neg x_3$: SAT $(\neg x_1 \wedge \neg x_2 \wedge x_3) \rightarrow$ **MCS : (C_1)**

  Adding : $\neg\neg y_1$, so $(\neg y_1 \vee x_1)$ reduces to x_1

  $x_1 \wedge \neg y_2 \wedge (\neg x_1 \vee x_2) \vee \neg x_2 \vee (\neg x_1 \vee x_3) \wedge \neg x_3$: UNSAT

  $x_1 \wedge \neg x_1 \wedge \neg y_3 \wedge \neg x_2$ ….: UNSAT

  …
- **K = 2**

  $\phi' = (\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3) \wedge y_1$

  $= x_1 \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3)$

  $= x_1 \wedge \neg y_2 \wedge \neg y_3 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$:UNSAT

  $x_1 \wedge \neg x_1 \wedge$…. :UNSAT

  …
- **K = 3**

  $x_1 \wedge \neg y_2 \wedge \neg y_3 \wedge \neg y_4 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$:UNSAT

  $x_1 \wedge \neg y_2 \wedge (\neg x_1 \vee x_2) \wedge \neg y_4 \wedge \neg y_5 \wedge \neg x_3$: SAT $(x_1, \neg y_2, x_2, \neg y_4, \neg y_5, \neg x_3)$: → **MCS : (C_2, C_4, C_5)**

  …