On Search Strategies for Constraint-Based Bounded Model Checking

Michel RUEHER

Joined work with Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini and Pascal Van Hentenryck

University Nice Sophia-Antipolis
I3S – CNRS, France

CP meets CAV

25 June – 29 June 2012
A CP framework for Bounded Program Verification

CPBPV, a Depth First Dynamic Exploration of the CFG

DPVS, a Dynamic Backjumping Strategy

The Flasher Manager Application

Discussion
Motivations

→ **Automatic generation of counterexamples**
violating a property on a limited model
of the program is very useful

→ **Challenge**: finding bugs for **realistic time periods**
for **real time applications**
Bounded program verification
(the array lengths, the variable values and the loops are bounded)

- **Constraint stores** to represent the specification and the program

- Program is partially correct if the **constraint store implies the post-conditions**

- Non deterministically exploration of execution paths
CP-based BMC mainly involves three steps:

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)
CP framework & BMC ...

- **CP framework**
  - Specification → constraints
  - Program → constraints *(on the fly)*
  - Solving Process
    - List of solvers tried in sequence on each selected node of the CFG
    - Takes advantage of the **structure** of the program

- **BMC based on SAT / SMT solvers**
  - Program & specification → Big Boolean formula
  - Solving Process
    - SAT solvers or SMT solvers have a "Global view"
    - Critical issue: **minimum conflict sets**
      (to limit backtracks & spurious solutions)
CP framework, pre-processing

Pre-processing

1. $P$ is unwound $k$ times $\rightarrow P_{uw}$

2. $P_{uw} \rightarrow DSA$, Dynamic Single Assignment form
   (each variable is assigned exactly once on each program path)

3. $DSA$ is simplified according to the specific property by applying slicing techniques

4. Domains of all variables are filtered by propagating constant values along the simplified CFG
A small example

```c
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; }  
}  

assert(c >= d + e); // property \(p_1\)
assert(f >= -b * e); // property \(p_2\)
```
A small example (continued)

Initial CFG

```c
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
}```
A small example (continued)

Simplified CFG

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

CP framework, language

- **Java** programs and **JML** specifications

  **JML** =
  
  - Comments in java code ("javadoc" like) (can be compiled and executed at run time)
  
  - Properties are directly expressed on the **program variables** → no need for abstraction
  
  - Pre-conditions and post-relations
  
  - **Exists** and **Forall** quantifiers

- **C** programs and **assertions**
CP framework, restrictions

- **Unit code** validation

- Data types: Booleans, integers, arrays of integers, [floats]

- **Bounded programs**: array lengths, number of unfoldings of loops, size of integers are known

- Normal behaviours of the method (no exception)

- JML specification:
  - post condition: the conjunction of use cases of the method
  - possibly a precondition
Building the constraint store: principle

- Each **expression** is mapped to a **constraint**: 
  \( \rho \) transforms program expressions into constraints

- SSA-like **variable renaming**: \( \sigma[v] \) is the current renaming of variable \( v \)

- **JML**:
  - \( \forall i \rightarrow \) conjunction of conditions
  - \( \exists i \rightarrow \) disjunction of conditions

  (\( i \) has bounded values)
Building the constraint store ...

**scalar assignment**

\[
\sigma_2 = \sigma_1[v/\sigma_1(v) + 1] \quad \& \quad c_2 \equiv (\rho \sigma_2 v) = (\rho \sigma_1 e)
\]

Program

\[
x = x + 1; \quad y = x \cdot y; \quad x = x + y;
\]

Constraints

\[
\{ x_1 = x_0 + 1, \quad y_1 = x_1 \cdot y_0, \quad x_2 = x_1 \cdot y_1 \}
\]
Building the constraint store ...

- **array assignment**

\[
\sigma_2 = \sigma_1[a/\sigma_1(a) + 1]
\]

\[
c_2 \equiv (\rho \; \sigma_2 \; a)[\rho \; \sigma_1 \; e_1] = (\rho \; \sigma_1 \; e_2)
\]

\[
c_3 \equiv \forall i \in 0..a.length(\rho \; \sigma_1 \; e_1) \neq i \rightarrow (\rho \; \sigma_2 \; a)[i] = (\rho \; \sigma_1 \; a)[i]
\]

\[
\langle [a[e_1] \leftarrow e_2, l], \sigma_1, c_1 \rangle \mapsto \langle [l], \sigma_2, c_1 \land c_2 \land c_3 \rangle
\]

**Program (a.length=8)**

\[
a[i] = x;
\]

**Constraints**

\[
\{ a_1[i_0] = x_0, i_0 \neq 0 \rightarrow a_1[0] = a_0[0],
\]

\[
i_0 \neq 1 \rightarrow a_1[1] = a_0[1], \ldots, i_0 \neq 7 \rightarrow a_1[7] = a_0[7]\}
\]

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

\[
a[i] = x\] is the **element constraint**: \(i\) and \(x\) are constrained variables whose values may be unknown
Building the constraint store ...

- **conditional instruction**: if \( b_i \); \( l \)

\[
\frac{c \land (\rho \sigma b) \text{ is satisfiable}}{\langle if \ b_i \ ; \ l, \sigma, c \rangle \rightarrow \langle i \ ; \ l, \sigma, c \land (\rho \sigma b) \rangle}
\]

\[
\frac{c \land \neg(\rho \sigma b) \text{ is satisfiable}}{\langle if \ b_i \ ; \ l, \sigma, c \rangle \rightarrow \langle l, \sigma, c \land \neg(\rho \sigma b) \rangle}
\]
Building the constraint store ... 

- **while instruction:** `while b i ; l`

\[
c \land (\rho \sigma b) \text{ is satisfiable} \\
\langle \text{while } b \ i ; \ l, \sigma, c \rangle \mapsto \langle i; \text{while } b \ i ; \ l, \sigma, c \land (\rho \sigma b) \rangle
\]

\[
c \land \neg(\rho \sigma b) \text{ is satisfiable} \\
\langle \text{while } b \ i; \ l, \sigma, c \rangle \mapsto \langle l, \sigma, c \land \neg(\rho \sigma b) \rangle
\]
CPBPV, Depth first exploration of the CFG

- Translate precondition of the specification (if it exists) into a set of constraints \texttt{PRECOND}

- Translate post condition of the specification into a set of constraints \texttt{POSTCOND}

- Explore each branch $B_i$ of the program and translate instructions of $B_i$ into a set of constraints \texttt{PROG\_Bi}
For each branch \( B_i \), solve \( \text{CSP}_i = \text{PROG}_B \land \text{PRECOND} \land \neg \text{POSTCOND} \)

- If for each branch \( B_i \) \( \text{CSP}_i \) is inconsistent, then the program is conform with its specification
- If for a branch \( B_i \) \( \text{CSP}_i \) has a solution, then this solution is a counterexample which illustrates a non-conformity

⚠️ Inconsistencies of \( \text{CSP}_i \) are detected at each node of the control flow graph
Current prototype – On the fly validation: if c then ... else ...

- If c can be simplified into constant value “true” or “false”, select the branch which corresponds to c

- If c is linear
  1. add decision c in linear_CSP
  2. solve linear_CSP
    - if linear_CSP has no solution, condition c is not feasible for the current path
      ~⇒ choose another path
    - if linear_CSP has a solution, we can’t conclude anything on complete_CSP
      ~⇒ investigate both branches c and ¬c
Current prototype – On the fly validation : if \( c \) then ... else ...

- If \( c \) is NOT linear :
  1. abstract decision \( c \) and add it in \( \text{boolean}_CSP \)
  2. solve \( \text{boolean}_CSP \)
     - \( \text{boolean}_CSP \) has no solution \( \Rightarrow \) choose another path
     - if \( \text{boolean}_CSP \) has a solution \( \Rightarrow \) investigate both branches \( c \) and \( \neg c \)

**Boolean abstraction**

- hash-table of decisions : keys are decisions, values are Boolean variables
- sub-expressions are shared \( \Rightarrow \) rewriting
Current prototype – On the fly validation: loops

Let $c$ be the entrance condition

- if $c$ is **trivially simplified** to “true” or “false”
  $\Rightarrow$ **enter** or **exit** the loop
- if $\{c + \text{linear}\_\text{CSP}\}$ is **inconsistent**
  $\Rightarrow$ add $\neg c$ to the CSPs and **exit** the loop

In other cases, unfold loop $\text{max}$ times:

- If $\text{max}$ is **reached**
  $\Rightarrow$ add $\neg c$ to the CSPs and **exit** the loop
- Else investigate **both** paths
Example: binary search (1)

```c
/*@ requires (\forall int i; i>=0 && i<t.length-1; t[i]<=t[i+1])
  @ ensures @ (\result!=-1 ==> t[\result] == v) &&
  @ (\result==-1 ==> \forall int k; 0<=k<t.length; t[k]!=v)
  @*/

1 static int binary_search(int[] t, int v)
2     int l = 0;
3     int u = t.length-1;
4     while (l <= u)
5         int m = (l + u) / 2;
6         if (t[m]==v) return m;
7         if (t[m] > v)
8             u = m - 1;
9         else
10             l = m + 1; // ERROR else u = m - 1;
11     return -1;
```
Example: binary search (2)

- **Precondition**

\[
\forall \text{int } i; i \geq 0 \\
& \& i < t.\text{length}-1; t[i] \leq t[i+1]
\]

\[
\text{CSP} \leftarrow t_0[0] \leq t_0[1] \land t_0[1] \leq t_0[2] \land \ldots \land t_0[6] \leq t_0[7]
\]

- **Initialization**

\[
\text{int } l = 0; \text{int } u = t.\text{length}-1;
\]

\[
\text{CSP} \leftarrow \text{CSP} \land l_0 = 0 \land u_0 = 7
\]
Example: binary search (2)

- **Precondition**

  \[
  \forall \text{int } i; i \geq 0 \\
  \text{&& } i < t.\text{length}-1; t[i] \leq t[i+1]
  \]

  \[
  \text{CSP} \leftarrow t_0[0] \leq t_0[1] \land t_0[1] \leq t_0[2] \land \ldots \land t_0[6] \leq t_0[7]
  \]

- **Initialization**

  \[
  \text{int } l = 0; \text{int } u = t.\text{length}-1;
  \]

  \[
  \text{CSP} \leftarrow \text{CSP} \land l_0 = 0 \land u_0 = 7
  \]
Example: binary search (3)

- **Loop**
  
  ```
  while (l<=u)
  ```

  **Enter into the loop since l₀ ≤ u₀ is consistent with the current constraint store**
  
  ```
  CSP ← CSP ∧ l₀ ≤ u₀
  ```

- **Assignment**
  
  ```
  int m=(l+u)/2;
  ```

  ```
  CSP ← CSP ∧ m₀ = (l₀ + u₀)/2 = 3
  ```
Example: binary search (3)

- Loop
  
  while (l<=u)

  Enter into the loop since $l_0 \leq u_0$ is consistent with the current constraint store
  
  $CSP \leftarrow CSP \land l_0 \leq u_0$

- Assignment
  
  int m=(l+u)/2;

  $CSP \leftarrow CSP \land m_0 = (l_0 + u_0)/2 = 3$
Example: binary search (4)

- **Conditional**

  
  ```c
  if (t[m] == v) return m;
  
  t_0[m_0] = v_0 \text{ is consistent with the constraint store so take the if part}
  
  CSP \leftarrow CSP \land t_0[m_0] = v_0
  ```

- **Complete execution path** \( p \) whose constraint store \( c_p \) is:

  
  \[ c_{pre} \land l_0 = 0 \land u_0 = 7 \land m_0 = 3 \land t_0[m_0] = v_0 \]
Example: binary search (4)

- **Conditional**
  
  ```
  if (t[m]==v) return m;
  
  t₀[m₀] = v₀ is consistent with the constraint store
  so take the if part
  CSP ← CSP ∧ t₀[m₀] = v₀
  ```

- **Complete execution path** $p$ whose **constraint store** $c_p$ is:
  
  $c_{pre} ∧ l₀ = 0 ∧ u₀ = 7 ∧ m₀ = 3 ∧ t₀[m₀] = v₀$
Example: binary search (5)

Return statement has been reached

▶ add negation of post condition and link JML \( \text{result} \) variable with returned value \( m_0 \)

\[
\text{result} != -1 \implies t[\text{result}] == v) \land \lnot (\text{result} == -1 \implies \forall \text{int } k; 0 \leq k < t.\text{length}; t[k] != v)
\]

\[
\begin{align*}
\lnot m_0 & = -1 \land t_0[m_0]! = v_0 \\
\lnot m_0 & = -1 \land (t_0[0] = v_0 \lor t_0[1] = v_0 \lor \ldots \lor t_0[6] = v_0)
\end{align*}
\]

▶ solve the CSP

There is No solution so the program is correct along this execution path

Go back to conditional if \( t[m] == v \) to explore the else part
Example: binary search (5)

Return statement has been reached

- add negation of post condition and link JML \( \text{\result} \) variable with returned value \( m_0 \)

\[
\text{\result} != -1 \implies \text{\ \text{\ \text{\text{t[\text{\result}] == v}) \&\&}} \\
(\text{\result} == -1 \implies \text{\forall int } k; \\
\quad 0 \leq k < \text{\text{\text{t.length; t[k]!}=v)}
\]

\[
\text{\\n
m_0! = -1 \land t_0[m_0]! = v_0 \lor} \\
m_0 = -1 \land (t_0[0] = v_0 \lor t_0[1] = v_0 \lor \ldots \lor t_0[6] = v_0)
\]

- **solve the CSP**
  - There is **No solution** so the program is **correct** along this execution path

**Go back** to conditional **if (t[m]==v)** to explore the **else** part
Implementation

- **Dedicated solvers**
  - **ad-hoc simplifier**: trivial simplifications and calculus on constants
  - **linear solver** (LP algorithm) + **MIP solver**
  - **Boolean solver** (SAT solver)
    (Boolean relaxation of the **non linear** constraints)
  - **CSP solver**: used if none of the other solver did find an inconsistency

- **Prototype**
  - Solvers: Ilog CPLEX11 and JSolver4verif
  - Written in **Java** using **JDT** (eclipse) for parsing Java programs

!! CPLEX is unsafe but Neumaier & Shcherbina
→ method for computing a certificate of infeasibility
Table: Results for a correct binary search program

<table>
<thead>
<tr>
<th>length</th>
<th>CPBPV</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.08s</td>
<td>1.37s</td>
</tr>
<tr>
<td>16</td>
<td>1.69s</td>
<td>1.43s</td>
</tr>
<tr>
<td>32</td>
<td>4.04s</td>
<td>KO</td>
</tr>
<tr>
<td>64</td>
<td>17.01s</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>136.80s</td>
<td></td>
</tr>
</tbody>
</table>

Table: Results for an incorrect binary search program

<table>
<thead>
<tr>
<th>length</th>
<th>CPBPV</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.027s</td>
<td>1.38s</td>
</tr>
<tr>
<td>16</td>
<td>0.037s</td>
<td>1.69s</td>
</tr>
<tr>
<td>32</td>
<td>0.064s</td>
<td>7.62s</td>
</tr>
<tr>
<td>64</td>
<td>0.115s</td>
<td>27.05s</td>
</tr>
<tr>
<td>128</td>
<td>0.241s</td>
<td>189.20s</td>
</tr>
</tbody>
</table>

!! CBMC only shows the decisions taken along the faulty path (they do not provide any value for the array nor the searched data)
Role of the different solvers

- **CPLEX, the MIP solver**, plays a key role

- There are only **length calls** to the CP solver (and much more calls to CPLEX)

- Almost **75% of the CPU time is spent in the CP solver**
Critical issues

- We do not need the Boolean abstraction to capture the control structure of the program
  → Use the CFG and constraints to prune the search space

- Depth first dynamic exploration of the CFG
  - Efficient if the variables are instantiated early
  - Blind searching: post-condition becomes active very late
DPVS, a Dynamic Backjumping Strategy

→ Generating *Counterexamples*

→ Starts from the postcondition and *jumps to the locations where the variables are assigned*
Why can we do it?

**Essential observation:**

When the program is in an SSA-like form, a path can be built in a non-sequential dynamic way.

CFG does not have to be explored in a top down (or bottom up) way: compatible blocks can just be collected in a non-deterministic way.
A Dynamic Backjumping Strategy

**DPVS** starts from the post-condition and dynamically collects program blocks which involve variables of the post-condition

**Why does it pay off?**

→ **Enforces the constraints** on the domains of the selected variables

→ **Detects inconsistencies earlier**
A small exemple

```c
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 $p_1$
assert(f >= -b * e); // property $p_2$
```
To prove property $p_1$, select node (12), then select node (4)

→ the condition in node (0) must be true

$S = \{c_1 < d_0 + e_0 \land c_1 = c_0 + d_0 + e_0 \land c_0 = a_0 \land a_0 \geq 0\}$

$= \{a_0 < 0 \land a_0 \geq 0\}$ ... inconsistent
Select node (8) → condition in node (0) must be false

\[ S = \{ c_1 < d_0 + e_0 \land c_1 = c_0 + d_0 + e_0 \land c_0 = b_0 \land a_0 < 0 \land d_0 = 1 \land e_0 = -a_0 \} \]

\[= \{ a_0 < 0 \land b_0 < 0 \} \]

**Solution** \( \{ a_0 = -1, b_0 = -1 \} \)
DPVS, pre-processing

Pre-processing

1. $P$ is **unwound** $k$ times $\rightarrow P_{uw}$

2. $P_{uw} \rightarrow DSA_{Puw}$, **Dynamic Single Assignment form** (each variable is assigned exactly once on each program path)

3. $DSA_{Puw}$ is **simplified according to the specific property** $prop$ by applying slicing techniques

4. Domains of all variables are filtered by **propagating constant values** along $G$, the simplified CFG
DPVS, Algorithm (scheme)

\[ S \leftarrow \text{negation of } \textit{prop} \ % \ \textit{constraint store} \]
\[ Q \leftarrow \text{variables in } \textit{prop} \ % \ \textit{queue of variables} \]

- While \( Q \neq \emptyset \), \( v \leftarrow \text{POP}(Q) \)
  - Search for a program block \( PB(v) \) where \( v \) is defined
    - PUSH(\( Q \), \( new\_var \)), \( new\_var = \text{new variables (\( \neq \) input variables)} \) of \( PB(v) \)
    - \( S \leftarrow S \cup \{ \text{definition of } v \ \text{and conditions required to reach definition of } v \} \)
  - IF \( S \) is inconsistent, backtrack & search another definition (otherwise the dual condition is cut off)

- IF \( Q = \emptyset \) search for an instantiation of the input variables (= counterexample)

If no solution exists, DPVS backtracks.
FM Application: Description of the module

- **A real time industrial application** from a car manufacturer (provided by Geensoft)

- **Flasher Manager (FM)**: controller that drives several functions related to the flashing lights

**Purpose:**

- to indicate a direction change
- to lock and unlock the car from the distance
- to activate the warning lights

- **Simulink model** of FM → C function $f_1$
FM Application: functionalities

- **Direction change**: Boolean input $R$ or $L$ rises from 0 to 1. The corresponding light then oscillates between on/off states with a period of **6 time-units** (e.g. 3 s) → output sequence of the form $[111000]$

- **Lock and unlock of the car**
  - If the unlock button is pressed while the car is unlocked, nothing shall happen.
  - If the unlock button is pressed while the car is locked, both lights shall flash with a **period of 2 time-units during 20 time-units** (fast flashes for a short time)
  - If the lock button is pressed while the car is unlocked, both lights shall go on for **10 time-units**, and then shall go off for another 10 time-units
  - If the lock button is pressed while the car is locked, both lights shall flash during **60 time-units with a period of 2 time-units** (fast flashes for a long time) ..

- **Warning function**: when the warning is on, both lights flash with a **period of 6 time-units**
FM Application: Simulink model(1)
FM Application: Simulink model (2)
Simulink model of FM → C function $f_1$

- 81 Boolean variables (6 inputs, 2 outputs) and 28 integer variables
- **300 lines of code**: nested conditionals including linear operations and constant assignments

Piece of code:

```c
and1_a=((Switch5==TRUE)&&(TRUE!=Unit_Delay3_a_DSTATE));
if ((TRUE==((and1_a[Unit_Delay_c_DSTATE])!= 0))) {
    rtb_Switch_b=0;
}
else {
    add_a = (1+Unit_Delay1_b_DSTATE);
    rtb_Switch_b = add_a;
}
superior_a = (rtb_Switch_b>=3);
```
FM Application: properties

\( p_1 \) The lights should never remain lit

\( p_2 \) The \texttt{Warning} function has priority over other flashing functions

\( p_3 \) When the warning button has been pushed and then released, the \texttt{Warning} function resumes to the \texttt{Flashers\_left} (or \texttt{Flashers\_right}) function, if this function was active when the warning button was pushed

\( p_4 \) When the \texttt{F} signal (for flasher active) is off, then the \texttt{Flashers\_left}, \texttt{Flashers\_right} and \texttt{Warning} functions are disabled. On the contrary, all the functions related to the lock and unlock of the car are maintained
FM Application: property $p_1$

- Property $p_1$: *The lights should never remain lit*

  Property $p_1$ concerns the behaviour of FM for an infinite time period

  $\rightarrow$ $p_1$ is violated when the lights remain on for $N$ consecutive time period

  $\rightarrow$ a loop (bounded by $N$) that counts the number of times where the output of FM has consecutively been true

  **Challenge:** bound $N$ as great as possible
**Program** under test for Property:

```c
void prop4(int d) {
    //number of time where the left light has been consecutively true
    int countL = 0;
    //number of time where the right light has been consecutively true
    int countR = 0;
    //consider d units of time
    for(int i=0;i<d;i++) {
        //non-deterministic values of the inputs
        L=nondet_in(); R=nondet_in();
        LK=nondet_in(); ULK=nondet_in();
        W=nondet_in(); F=nondet_in();
        //call to fl() to simulate one pass through the module
        fl();
        if (outL)
            //the left light has been consecutively true one more time
            countL++;
        else
            //the left light has not been consecutively true
            countL=0;
        if (outR)
            //the right light has been consecutively true one more time
            countR++;
        else
            //the right light has not been consecutively true
            countR=0;
    }
    //if countL and countR are less than d,
    //then the lights did not remain lit
    assert (countL<d && countR<d);
}
```
Experiments: tools

- **DPVS**, implemented in **Comet**, a hybrid optimization platform for solving combinatorial problems

- **CPBPV***, an optimized version of CPBPV based on a dynamic **top down strategy**

- **CBMC**, one of the best bounded model checkers

Experiments were performed on a Quad-core Intel Xeon X5460 3.16GHz clocked with 16Gb memory. All times are given in seconds.
Experiments (property p1)

Solving time:

<table>
<thead>
<tr>
<th>N</th>
<th>CBMC</th>
<th>DPVS</th>
<th>CPBPV*</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.03</td>
<td>0.02</td>
<td>0.84</td>
</tr>
<tr>
<td>100</td>
<td>58.52</td>
<td>1.11</td>
<td>TO</td>
</tr>
<tr>
<td>200</td>
<td>232.19</td>
<td>1.7</td>
<td>TO</td>
</tr>
<tr>
<td>400</td>
<td>TO</td>
<td>3.83</td>
<td>TO</td>
</tr>
<tr>
<td>800</td>
<td>TO</td>
<td>9.35</td>
<td>TO</td>
</tr>
<tr>
<td>1600</td>
<td>TO</td>
<td>26.2</td>
<td>TO</td>
</tr>
</tbody>
</table>

Presolving time:

<table>
<thead>
<tr>
<th>N</th>
<th>CBMC</th>
<th>DPVS &amp; CPBPV*</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.366</td>
<td>0.48</td>
</tr>
<tr>
<td>100</td>
<td>96.21</td>
<td>14.95</td>
</tr>
<tr>
<td>200</td>
<td>395.46</td>
<td>21.65</td>
</tr>
<tr>
<td>400</td>
<td>TO</td>
<td>83.81</td>
</tr>
<tr>
<td>800</td>
<td>TO</td>
<td>218.15</td>
</tr>
<tr>
<td>1600</td>
<td>TO</td>
<td>531.82</td>
</tr>
</tbody>
</table>
Presolving, search, and total times in seconds for checking Property $p_2$ with 10 unfoldings

<table>
<thead>
<tr>
<th>Tool</th>
<th>Presolving</th>
<th>Search</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBMC</td>
<td>0.89</td>
<td>0.23</td>
<td>1.12</td>
</tr>
<tr>
<td>CBMC$_{z3}$</td>
<td>0.85</td>
<td>2.7</td>
<td>3.55</td>
</tr>
<tr>
<td>DPVS</td>
<td>3.89</td>
<td>0.08</td>
<td>3.97</td>
</tr>
<tr>
<td>DPVS$_{z3}$</td>
<td>0.34</td>
<td></td>
<td>4.23</td>
</tr>
</tbody>
</table>

This property does not hold (only 3 unfoldings are required)

Property 3 and 4 couldn’t be checked
### Discussion

#### Experiments on the binary search

<table>
<thead>
<tr>
<th>Length</th>
<th>CBMC</th>
<th>DPVS</th>
<th>CPBPV*</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5.732</td>
<td>0.529</td>
<td>0.107</td>
</tr>
<tr>
<td>8</td>
<td>110.081</td>
<td>35.074</td>
<td>0.298</td>
</tr>
<tr>
<td>16</td>
<td>TO</td>
<td>TO</td>
<td>1.149</td>
</tr>
<tr>
<td>64</td>
<td>TO</td>
<td>TO</td>
<td>27.714</td>
</tr>
<tr>
<td>128</td>
<td>TO</td>
<td>TO</td>
<td>153.646</td>
</tr>
</tbody>
</table>

- DPVS and CBMC waste a lot of time in exploring the different paths
- CPBPV* incrementally adds the decisions taken along a path → well adapted for the Binary Search program

**On going work**: Combining strategies