Hybrid Constraint-Based Bounded Program Verification

Michel RUEHER

University of Nice Sophia-Antipolis / I3S – CNRS, France

Courtesy to Hélène COLLAVIZZA, Nguyen Le VINH and Pascal Van HENTENRYCK

June, 2011

ACP Summer School
“Hybrid Methods for Constraint Programming”
Turunç

Outline

Basics on Bounded Model Checking (BMC)

A CP framework for Bounded Program Verification

CPBPV, a Depth First Dynamic Exploration of the CFG

DPVS

The Flasher Manager Application

Discussion

Basics on BMC

- **Mechanically check properties of models**
- Widely used in **hardware verification and software verification**
- Automatic generation of **counterexamples**

BMC: key features

- **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

Algorithm for Model Checking Safety

% set of states: S, initial states: I, transition relation: T
% **bad states B reachable from I via T?**

```
bounded_model_checker$_{forward}(I, T, B, k)$
    $S_C = \emptyset; S_N = I; n = 1$
    while $S_C \neq S_N$ and $n < k$ do
       if $B \cap S_N \neq \emptyset$
          then return "found error trace to bad states";
       else $S_C = S_N$;
            $S_N = S_C \cup T(S_C)$;
            $n = n + 1$;
     done
    return "no bad state reachable";
```

BMC framework

BMC: Bounded Model Checking

- BMC: falsification of a given property is checked for a given **bound**
- BMC mainly involves three steps:
  1. the **program is unwound k** times,
  2. the unwound program and the property are translated into **a big propositional formula** ϕ
     ϕ **is satisfiable iff there exists a counterexample of depth less than** k
  3. **A SAT-solver or SMT-solver** is used for checking the satisfiability of ϕ

CP framework

- **A CP framework for Bounded Program Verification**

Overall view of CP framework

- **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 ...

CP-based Bounded Program Verification

- CP-based BMC: falsification of a given property is checked for a given **bound**
- 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 (SAT solvers
    & specialised solvers)
    ⇝ **spurious solutions** → **backtracks**
    → Critical issue: **minimum conflict sets**

CP framework, pre-processing

Pre-processing

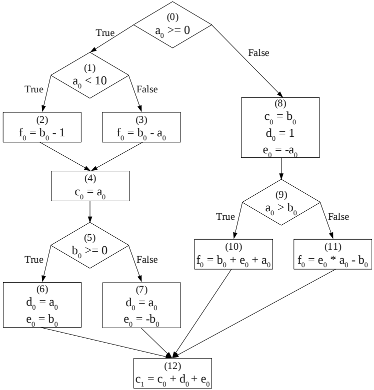
1. P is **unwound k times** $\rightarrow$ P_{uw}
2. $P_{uw} \rightarrow DSA_{P_{uw}}$, **Dynamic Single Assignment form** (each variable is assigned exactly once on each program path)
3. $DSA_{P_{uw}}$ is **simplified according to the specific property** *prop* by applying slicing techniques
4. Domains of all variables are filtered by **propagating constant values** along G, the simplified CFG

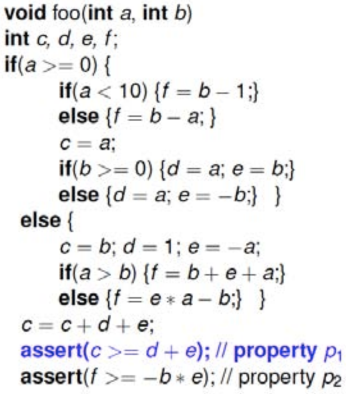
A small example

```
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
```

A small example(continued)

Initial CFG

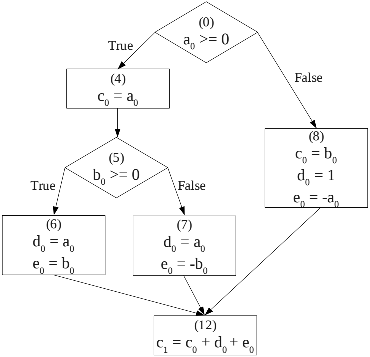


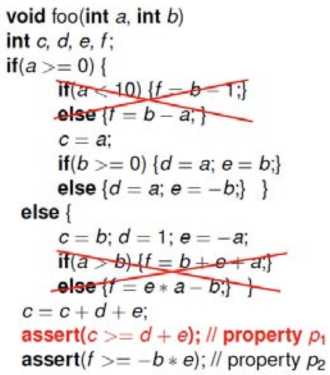


```
void foo(int a, int b)
int c, d, e, f;
if(a >= 0) {
      if(a < 10) {f = b - 1;}
      else {f = b - a; }
      c = a;
      if(b >= 0) {d = a; e = b;}
      else {d = a; e = -b;}  }
 else {
      c = b; d = 1; e = -a;
      if(a > b) {f = b + e + a;}
      else {f = e * a - b;}  }
 c = c + d +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

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 : integers, arrays of integers
- **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**: ρ transforms program expressions into constraints
- SSA-like **variable renaming**: $\sigma[\mathbf{v}]$ is the current renaming of variable **v**
- JML :
  - **forall i** $\longrightarrow$ conjunction of conditions
  - **exist i** $\longrightarrow$ disjunction of conditions

  (**i** has bounded values)

Building the constraint store ...

- **scalar assignment**

$$\frac{\sigma_2 = \sigma_1[v/\sigma_1(v)+1] \ \ \& \ \ c_2 \equiv (\rho\,\sigma_2\ v) = (\rho\,\sigma_1\ e)}{\langle [v \leftarrow e\ ,\ l], \sigma_1, c_1 \rangle \longmapsto \langle [l], \sigma_2, c_1 \wedge c_2 \rangle}$$

Program

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

Constraints

$\{x_1 = x_0 + 1, y_1 = x_1 * y_0, x_2 = x_1 * y_1\}$

Building the constraint store ...

- **array assignment**

$$\frac{\begin{array}{l}\sigma_2 = \sigma_1[a/\sigma_1(a)+1] \\ c_2 \equiv (\rho\ \sigma_2\ a)[\rho\ \sigma_1\ e_1] = (\rho\ \sigma_1\ e_2) \\ c_3 \equiv \forall i \in 0..a.length(\rho\ \sigma_1\ e_1) \neq i \rightarrow (\rho\ \sigma_2\ a)[i] = (\rho\ \sigma_1\ a)[i]\end{array}}{\langle [a[e_1] \leftarrow e_2, l], \sigma_1\ , c_1\rangle \longmapsto \langle [l], \sigma_2, c_1 \wedge c_2 \wedge c_3\rangle}$$

Program (a.length=8)

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

Constraints

$\{a_1[i_0] = x_0, i_0 \neq 0 \rightarrow a_1[0] = a_0[0],$
$i_0 \neq 1 \rightarrow a_1[1] = a_0[1], ..., i_0 \neq 7 \rightarrow a_1[7] = a_0[7]\}$

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**

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i\ ; l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

$$\frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

Building the constraint store ...

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

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i; \mathit{while}\ b\ i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

$$\frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i; l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

Depth first search strategies

- **CPBPV**, Depth first exploration of the CFG

CPBPV, Overall view

- Translate precondition of the specification (if it exists) into a set of constraints **PRECOND**
- Translate post condition of the specification into a set of constraints **POSTCOND**
- Explore **each branch** B_i of the program and translate instructions of B_i into a set of constraints **PROG_Bi**

CPBPV, the validation process

- For each branch B_i, solve **CSPi = PROG_Bi** $\wedge$ **PRECOND** $\wedge$ **NOT(POSTCOND)**
  - If for each branch B_i **CSPi is inconsistent** , then the program is **conform** with its specification
  - If for a branch B_i **CSPi has a solution** , then this solution is a **test case** which illustrates a **non-conformity**

⚠ **Inconsistencies of CSPi** are detected at **each node** of the control flow graph

Binary search (1)

```
/*@ 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)
@*/

static int binary_search(int[] t, int v)
     int l = 0;
     int u = t.length-1;
     while (l <= u)
          int m = (l + u) / 2;
          if (t[m]==v)   return m;
          if (t[m] > v)
                u = m - 1;
          else
                l = m + 1;  // ERROR else u = m - 1;
   return -1;
```

Binary search (2)

- **Precondition**

```
\forall int i;i>=0
               && i<t.length-1;t[i]<=t[i+1]
```

$\mathbf{CSP} \leftarrow \mathbf{t_0[0]} \le \mathbf{t_0[1]} \wedge \mathbf{t_0[1]} \le \mathbf{t_0[2]} \wedge \ldots \wedge \mathbf{t_0[6]} \le \mathbf{t_0[7]}$

- **Initialization**

```
 int l=0;int u=t.length-1;
```

$\mathbf{CSP} \leftarrow \mathbf{CSP} \wedge \mathbf{l_0} = \mathbf{0} \wedge \mathbf{u_0} = \mathbf{7}$

Binary search (2)

- **Precondition**

```
\forall int i;i>=0
              && i<t.length-1;t[i]<=t[i+1]
```

$\mathrm{CSP} \leftarrow t_0[0] \leq t_0[1] \wedge t_0[1] \leq t_0[2] \wedge \ldots \wedge t_0[6] \leq t_0[7]$

- **Initialization**

```
int l=0;int u=t.length-1;
```

$\mathrm{CSP} \leftarrow \mathrm{CSP} \wedge l_0 = 0 \wedge u_0 = 7$

Binary search (3)

- **Loop**

  `while (l<=u)`

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

- **Assignment**

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

  $CSP \leftarrow CSP \wedge m_0 = (l_0 + u_0)/2 = 3$

Binary search (3)

- **Loop**

  ```
  while (l<=u)
  ```

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

  $CSP \leftarrow CSP \wedge l_0 \leq u_0$

- **Assignment**

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

  $CSP \leftarrow CSP \wedge m_0 = (l_0 + u_0)/2 = 3$

Binary search (4)

- **Conditional**

  `if (t[m]==v) return m;`

  $t_0[m_0] = v_0$ is consistent with the constraint store so take the if part
  $CSP \leftarrow CSP \wedge t_0[m_0] = v_0$

- **Complete execution path** p whose **constraint store** c_p is:
  $c_{pre} \wedge l_0 = 0 \wedge u_0 = 7 \wedge m_0 = 3 \wedge t_0[m_0] = v_0$

Binary search (4)

- **Conditional**

  `if (t[m]==v) return m;`

  $t_0[m_0] = v_0$ is consistent with the constraint store so take the if part
  $CSP \leftarrow CSP \wedge t_0[m_0] = v_0$

- **Complete execution path** *p* whose **constraint store** c_p is:
  $c_{pre} \wedge l_0 = 0 \wedge u_0 = 7 \wedge m_0 = 3 \wedge t_0[m_0] = v_0$

Binary search (5)

Return statement has been reached

- add negation of post condition and link JML \result variable with returned value m_0

```
\result!=-1 ==> t[\result] == v) &&
(\result==-1 ==> \forall int k;
                0<=k<t.length; t[k]!=v)
```

$\backslash m_0 ! = -1 \land t_0[m_0] ! = v_0 \lor$

$\backslash m_0 = -1 \land (t_0[0] = v_0 \lor t_0[1] = v_0 \lor \ldots \lor t_0[6] = v_0)$

- **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

Binary search (5)

Return statement has been reached

- add negation of post condition and link JML \result variable with returned value m_0

```
\result!=-1 ==> t[\result] == v) &&
(\result==-1 ==> \forall int k;
                 0<=k<t.length; t[k]!=v)
```

$$\backslash \mathbf{m_0} ! = -\mathbf{1} \wedge \mathbf{t_0}[\mathbf{m_0}] ! = \mathbf{v_0} \vee$$
$$\backslash \mathbf{m_0} = -\mathbf{1} \wedge (\mathbf{t_0}[0] = \mathbf{v_0} \vee \mathbf{t_0}[1] = \mathbf{v_0} \vee \ldots \vee \mathbf{t_0}[6] = \mathbf{v_0})$$

- **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

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
       $\rightsquigarrow$ **choose another path**
     - if **linear_CSP** **has a solution**, we can’t conclude anything on **complete_CSP**
       $\rightsquigarrow$ **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 **boolean_CSP**
  2. solve **boolean_CSP**
     - **boolean_CSP** **has no solution** ⇝ choose another path
     - if **boolean_CSP** **has a solution** ⇝ investigate both branches **c** and **¬c**

Boolean abstraction

- hash-table of decisions : keys are decisions, values are Boolean variables
- sub-expressions are shared → rewriting

Current prototype – On the fly validation : loops

Let **c** be the entrance condition

- if **c** is **trivially simplified** to “true” or “false”
  $\rightsquigarrow$ **enter** or **exit** the loop
- if {**c** $+$ **linear_CSP** } is **inconsistent**
  $\rightsquigarrow$ add $\neg$**c** to the CSPs and **exit** the loop

In other cases, unfold loop **max** times:

- If **max** is **reached**
  $\rightsquigarrow$ add $\neg$**c** to the CSPs and **exit** the loop
- Else investigate **both** paths

Experiments

We compared CPBVP with the following frameworks:

- **ESC/Java**, an Extended Static Checker for Java
  ⇝ run-time errors in JML-annotated Java programs (static analysis of the code and its annotations)
- **CBMC**, a Bounded Model Checker for ANSI-C and C++ programs
  ⇝ verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions
- **BLAST**, a software model checker for C program (Berkeley Lazy Abstraction Software Verification Tool)
- **EUREKA**, a C bounded model checker which uses an SMT solver instead of an SAT solver
- **Why**, a verification platform which integrates provers (proof assistants such as Coq, PVS, HOL 4,...) and decision procedures (Simplify, Yices, ...)

Binary search

	length	8	16	32	64	128
CPBPV	**time**	1.08s	1.69s	4.04s	17.01s	136.80s
CBMC	**time**	1.37s	1.43s	KO		
Why	**inv**	11.18s				
	–	KO				
ESC/Java	Error					
BLAST	KO					

- *EUREKA* **tool** : cannot handle because of expression $\mathbf{m} = (\mathbf{u} + \mathbf{l})/\mathbf{2}$
- **CP execution paths** explored given by the recurrence relation: $\mathbf{P}(\mathbf{2}) = \mathbf{P}(\mathbf{4})$; $\mathbf{P}(\mathbf{2n}) = \mathbf{2P}(\mathbf{n}) + \log(\mathbf{n})$

length	CPBPV	ESC/Java	CBMC	WHY inv	BLAST
8	0.027s	1.21 s	1.38s	KO	KO
16	0.037s	1.347 s	1.69s	KO	KO
32	0.064s	1.792 s	7.62s	KO	KO
64	0.115s	1.886 s	27.05s	KO	KO
128	0.241s	1.964 s	189.20s	KO	KO

Table: Experimental Results for an Incorrect Binary Search

- **CBMC and ESC/Java only show the decisions taken along the faulty path (they do not provide any value for the array nor the searched data)**

Binary search

	length	8	16	32	64	128
CPBPV	**time**	1.08s	1.69s	4.04s	17.01s	136.80s
CBMC	**time**	1.37s	1.43s	KO		
Why	**inv**	11.18s				
	–	KO				
ESC/Java	Error					
BLAST	KO					

- *EUREKA* **tool** : cannot handle because of expression $\mathbf{m} = (\mathbf{u} + \mathbf{l})/\mathbf{2}$
- **CP execution paths** explored given by the recurrence relation: $\mathbf{P}(\mathbf{2}) = \mathbf{P}(\mathbf{4});\quad \mathbf{P}(\mathbf{2n}) = \mathbf{2P}(\mathbf{n}) + \log(\mathbf{n})$

length	CPBPV	ESC/Java	CBMC	WHY inv	BLAST
8	0.027s	1.21 s	1.38s	KO	KO
16	0.037s	1.347 s	1.69s	KO	KO
32	0.064s	1.792 s	7.62s	KO	KO
64	0.115s	1.886 s	27.05s	KO	KO
128	0.241s	1.964 s	189.20s	KO	KO

Table: Experimental Results for an Incorrect Binary Search

- **CBMC and ESC/Java only show the decisions taken along the faulty path (they do not provide any value for the array nor the searched data)**

Tritype

Takes 3 integers (triangle sides) and returns the type of triangle

- CP :**10 paths explored among 57** – correspond to actual inputs because of complex conditionals
- CP and Why : time does not depend on the size of the integers
- earlier approach (Boolean abstraction, TACAS'06): 8.52s for integers coded on 16 bits, 92 spurious paths

	CPBPV	**ESC/Java**	**CBMC**	**Why**	**BLAST**
time	**0.287s**	**1.828s**	**0.82s**	**8.85s**	**KO**

Sum of squares

```
/*@ requires (n == t.length-1)
  @   & (\forall int i; i>=0 & i<tab.length;
  @                         (0<=t[i] & t[i]<=n)
  @   & (\alldifferent t)
  @ ensures \result == n*(n+1)*(2*n+1)/6 @*/
  int sum(int[] t, int n)
        int s = 0;
        int i = 0;
        while (i!=t.length)
               s=s+t[i]*t[i]
               i =i+1;
        return s;
```

- Using global constraint **alldiff**
- Solving **non linear** problems
- $66.179s$ for $n = 10$

Role of the different solvers

- **CPLEX, the MIP solver**, plays a key role in all these benchmarks:
  - **Tritype:** the CP solver is never called
  - **Binary search:** there are only length calls to the CP solver (and much more calls to CPLEX) but almost 75% of the CPU time is spent in the CP solver
  - **Sum of squares:** 80% 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 Constraint-Based BMC Strategy For Generating Counterexamples

Motivations

Formal proof methods that ensure the ***absence of all bugs*** are **too expensive**, or require manual efforts

→ **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**

Key points of DPVS

A **new search strategy** for verifying a restricted class of Java or C programs:

→ **Non sequential dynamic exploration of the CFG**

- **CPBPV: Depth first dynamic exploration** of the CFG

  → *Postcondition is used very late* because of the variables renaming

- **DPVS: Non-sequential exploration** of the CFG

  → *Starts from the postcondition and jumps to the locations where the variables are assigned*

Non sequential dynamic constraint based exploration strategy

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

Non sequential dynamic constraint based exploration strategy

Why does it pay off

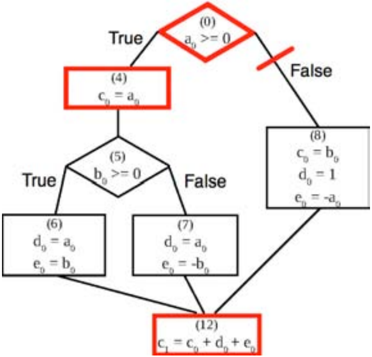
- **DPVS** **starts from the post-condition** and dynamically collects program blocks which involve **variables of the post-condition**
- Collecting as much information as possible on a given variable

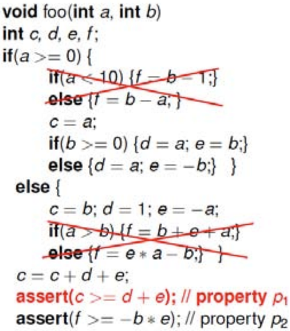
  → **enforces the constraints on its domain and reduces the search space**
- **Constraint solving is integrated with state exploration** to prune the state space as early as possible

A small exemple

```
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 exemple(continued)





```
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
```

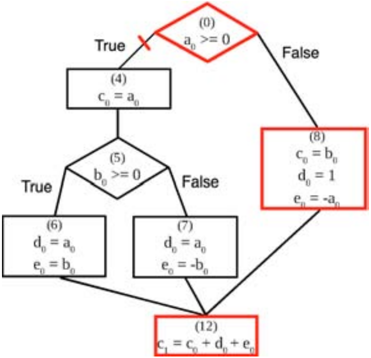
To prove property p_1, select node (12), then select node (4)

$\rightarrow$ the condition in node (0) must be true

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

$= \{a_0 < 0 \wedge a_0 \geq 0\}$ **... inconsistent**

A small exemple(continued)



Select node (8) $\rightarrow$ condition in node (0) must be false

$$S = \{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0$$
$$\wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0\}$$
$$= \{a_0 < 0 \wedge b_0 < 0\}$$

Solution $\{a_0 = -1, b_0 = -1\}$

DPVS, pre-processing

Pre-processing

1. P is **unwound k times** $\rightarrow$ P_{uw}
2. $P_{uw} \rightarrow DSA_{P_{uw}}$, **Dynamic Single Assignment form** (each variable is assigned exactly once on each program path)
3. $DSA_{P_{uw}}$ 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)

$S \leftarrow$ negation of *prop % constraint store*
$Q \leftarrow$ variables in *prop % queue of variables*

- `While` $Q \neq \emptyset, v \leftarrow$ POP(Q)
    - **Search for a program block $PB(v)$ where v is defined**
      PUSH(Q, new_var), $new\_var =$ new variables ($\neq$ input variables) of $PB(v)$
      $S \leftarrow S \cup$ {definition of v 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.

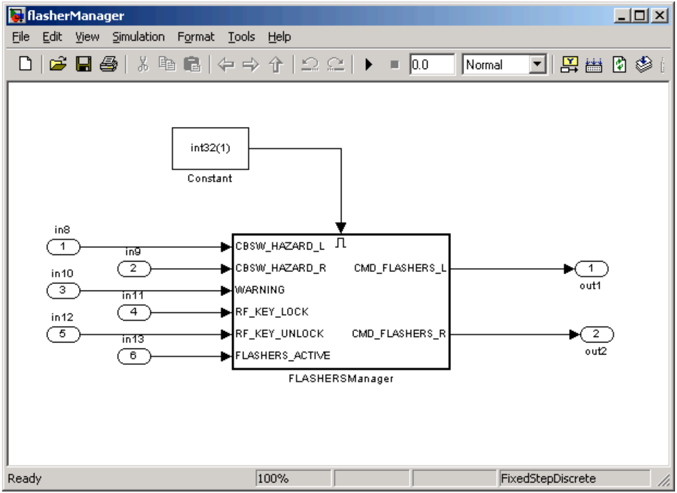
FM Application: Description of the module

- **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

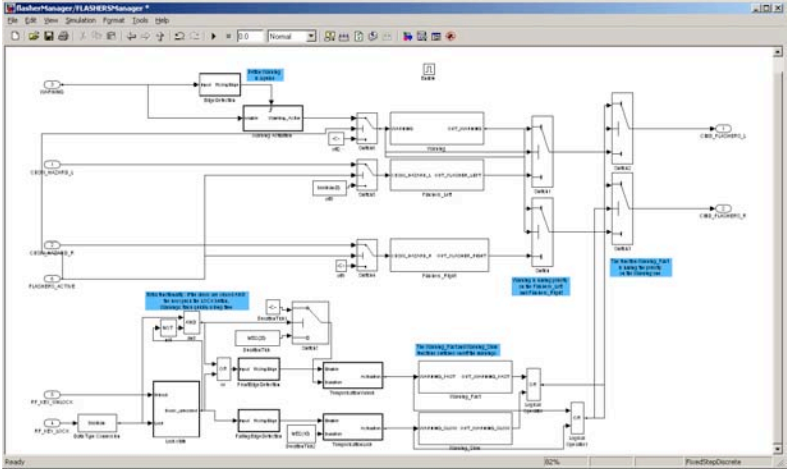
  Purpose:

  - 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: Simulink model(1)



FM Application: Simulink model (2)

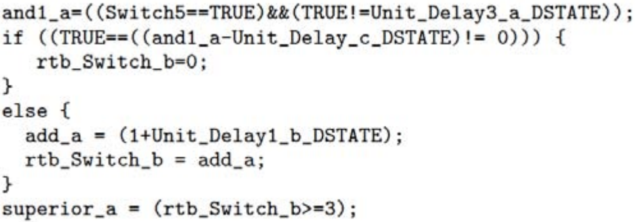


FM Application: Function f_1

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:



```
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 When the warning button has been pushed and then released, the `Warning` function resumes to the `Flashers_left` (or `Flashers_right`) function, if this function was active when the warning button was pushed
- p_3 When the `F` signal (for flasher active) is off, then the `Flashers_left`, `Flashers_right` and `Warning` functions are desabled. On the contrary, all the functions related to the lock and unlock of the car are maintained
- p_4 The `Warning` function has priority over other flashing functions

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**

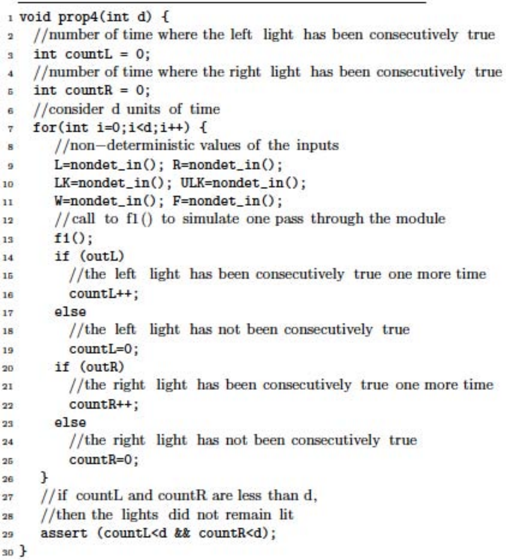
  → p_1 is violated when the lights remain on for *N* **consecutive time period**

  → 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**

FM Application: property p_1

Program under test for Property:



```
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 f1() to simulate one pass through the module
     f1();
     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 (results)

Solving time:

N	CBMC	DPVS	CPBPV*
5	0.03	**0.02**	0.84
100	57.27	**1.95**	TO
200	232.19	**3.45**	TO
400	TO	**4.66**	TO

Pre-processing time:

N	CBMC	DPVS	CPBPV*
5	0.366	**0.480**	0.480
100	65.190	**9.750**	9.750
200	395.46	**21.65**	21.65
400	TO	**50.90**	50.90

Discussion

Experiments on the binary search

Length	CBMC	DPVS	CPBPV*
4	5.732	0.529	**0.107**
8	110.081	35.074	**0.298**
16	TO	TO	**1.149**
64	TO	TO	**27.714**
128	TO	TO	**153.646**

- **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

Discussion (continued)

- **Combining strategies**
- Using counter examples for **errors localization**