Recherche de Bugs sous Contrainte

Hélène Collavizza\textsuperscript{1}, Michel Rueher\textsuperscript{1}, Pascal Van Hentenryck\textsuperscript{2}

\textsuperscript{1}University of Nice Sophia-Antipolis / CNRS, France
\textsuperscript{2}Brown University, Providence, USA

Journée du Pôle MDSC, Juin 09
A novel CP framework for bounded program verification (CPBPV)

- **CP framework**
  - **Constraint stores** to represent the specification and the program
  - **Nondeterministically** exploration of execution paths

- **Bounded program**
  - the array lengths, the variable values and the loops are **bounded**

- **Program verification**
  - A program is partially correct if the **constraint store implies the post-conditions**
Essential observation

- Verifying the conformity between a program and its specification
  → to check all executables paths
  (explicitly or implicitly)

⚠️ Not the case in model-checking tools
  ⇝ to detect violations of specific properties
  (safety, liveness properties, ...)

• Verifying the conformity between a program and its specification
→ to check all executables paths (explicitly or implicitly)
Benefits of CPBPV

- No predicate abstraction
  → no spurious paths

- Early pruning of execution paths
  (incremental detecting inconsistent constraint stores)

- Rich language of CP
  combination of constraints, element constraint, ...

- Parametrized with a list of solvers tried in sequence
CPBPV versus SAT/SMT based BMC

- **CPBPV**
  - Specification $\rightarrow$ constraints
    - Program – on the fly $\rightarrow$ constraints
  - Solving Process
    - List of solvers on each branch
      - Takes advantage of the structure of the program
      - No global view

- **SAT/SMT based BMC**
  - Program + specification $\rightarrow$ Big boolean formula
  - Solving Process
    - SAT solvers or SMT solvers
      - Global view
      - Spurious solutions
      - Structure of the program is lost
Langage

- **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
Current 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:
  - postcondition: the conjunction of use cases of the method
  - possibly a precondition
The validation process (1)

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

- Translate postcondition of the specification into a set of constraints \textbf{POSTCOND}

- Explore each branch $B_i$ of the program and translate instructions of $B_i$ into a set of constraints \textbf{PROG}_{Bi}
The validation process (2)

- For each branch $B_i$, solve $\text{CSP}_i = \text{PROG}_{B_i} \land \text{PRECOND} \land \neg \text{POSTCOND}$

  - If for each branch, CSPi is inconsistent, then the program is conform with its specification

  - If for some 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
Building the constraint store: principle

- Each **expression** is mapped to a **constraint**: $ho$ 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

\[
\sigma_2 = \sigma_1[v/\sigma_1(v) + 1] \quad \& \quad c_2 \equiv (\rho \sigma_2 v) = (\rho \sigma_1 e) \\
\langle [v \leftarrow e, l], \sigma_1, c_1 \rangle \mapsto \langle [l], \sigma_2, c_1 \land c_2 \rangle
\]

Program

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

Constraints

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

**array assignment**

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

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

\[c_3 \equiv \forall i \in 0..\text{a.length}(\rho \sigma_1 e_1) \neq i \Rightarrow (\rho \sigma_2 a)[i] = (\rho \sigma_1 a)[i]\]

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

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

\[i = \text{a}[1]; \text{a}[i] = x;\]

**Constraints**

\[\{i_0 = a_0[1], a_1[i_0] = x_0, i_0 \neq 0 \Rightarrow a_1[0] = a_0[0], i_0 \neq 1 \Rightarrow a_1[1] = a_0[1], \ldots, i_0 \neq 7 \Rightarrow a_1[7] = a_0[7]\}\]

\[\text{guard} \Rightarrow \text{body is a guarded constraint}\]

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

- conditional instruction

\[
\begin{align*}
\langle [ \text{if } b \ i , \ l ] , \sigma , c \rangle & \mapsto \langle [ i , l ] , \sigma , c \land ( \rho \sigma b ) \rangle \\
\langle [ \text{if } b \ i , \ l ] , \sigma , c \rangle & \mapsto \langle [ l ] , \sigma , c \land \neg ( \rho \sigma b ) \rangle
\end{align*}
\]
Building the constraint store ...

- **while instruction**

\[
\begin{align*}
\text{if } c \land (\rho \sigma b) \text{ is satisfiable} & \quad \langle [\text{while } b \ i \ , \ l], \sigma, c \rangle \rightarrow \langle [i, \text{while } b \ i \ , \ l], \sigma, c \land (\rho \sigma b) \rangle \\
\text{if } c \land \neg(\rho \sigma b) \text{ is satisfiable} & \quad \langle [\text{while } b \ i \ , \ l], \sigma, c \rangle \rightarrow \langle [l], \sigma, c \land \neg(\rho \sigma b) \rangle
\end{align*}
\]
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;
```
Binary search (2)

- **Precondition**

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

  \[
  \text{CSP} \leftarrow t^0[0] \leq t^0[1] \& t^0[1] \leq t^0[2] \& \ldots \& t^0[6] \leq t^0[7]
  \]

- **Initialization**

  \[
  \text{int } l = 0; \text{int } u = t.\text{length}-1; \\
  \text{CSP} \leftarrow \text{CSP} \& l^0 = 0 \& u^0 = 7
  \]
## Binary search (2)

- **Precondition**

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

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

- **Initialization**

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

  \[
  \text{CSP} \leftarrow \text{CSP} \land l^0 = 0 \land u^0 = 7
  \]
Binary search (3)

- **Loop**

  while (l<=u)

  Enter into the loop since $l^0 \leq u^0$ is consistent with the current constraint store

  $\text{CSP} \leftarrow \text{CSP} \land l^0 \leq u^0$

- **Assignement**

  int m=(l+u)/2;

  $\text{CSP} \leftarrow \text{CSP} \land m^0 = (l^0 + u^0)/2 = 3$
Binary search (3)

- **Loop**

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

  Enter into the loop since \( l^0 \leq u^0 \) is consistent with the current constraint store

  \[
  \text{CSP} \leftarrow \text{CSP} \land l^0 \leq u^0
  \]

- **Assignment**

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

  \[
  \text{CSP} \leftarrow \text{CSP} \land m^0 = \frac{l^0 + u^0}{2} = 3
  \]
Binary search (4)

- **Conditional**

  \[
  \text{if} \ (t[m]==v) \ \text{return} \ m;
  \]

  \(t^0[m^0] = v^0\) 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:

  \[
  c_{\text{pre}} \land l^0 = 0 \land u^0 = 7 \land m^0 = 3 \land t^0[m^0] = v^0
  \]
Binary search (4)

- **Conditional**

  \[
  \text{if } (t[m]==v) \text{ return } m;
  \]

  \(t^0[m^0] = v^0\) is consistent with the constraint store so take the if part

  \[
  \text{CSP } \leftarrow \text{CSP } \land t^0[m^0] = v^0
  \]

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

  \[
  c_{\text{pre}} \land l^0 = 0 \land u^0 = 7 \land m^0 = 3 \land t^0[m^0] = v^0
  \]
Binary search (5)

Return statement has been reached

- add negation of postcondition and link JML $\verb|esult|! \neq -1$ with returned value $m^0$

$$\verb|esult|! \neq -1 \implies t[\verb|esult|] = v) \land (\verb|esult| == -1 \implies \forall \text{int } k; 0 \leq k < t.\text{length}; t[k] \neq v)$$

$$\neg m^0 = -1 \land t^0[m^0] = v_0 \lor \neg m^0 = -1 \land (t^0[0] = v_0 \lor t^0[1] = v_0 \lor ... \lor t^0[6] = v_0)$$

- 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 postcondition and link JML \( \text{\textbackslash result} \) variable with returned value \( m^0 \)

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

\[
m^0 = -1 \land t^0[m^0] != v^0 \lor
m^0 = -1 \land (t^0[0] = v^0 \lor t^0[1] = v^0 \lor \ldots \lor t^0[6] = v^0)
\]

- 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

- **First Prototype**
  - *Written in JAVA* by Hélène Collavizza
  - Solvers: Ilog CPLEX11 and JSolver4verif
  - Written in *Java* using *JDT* (eclipse) for parsing Java programs
First 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, otherwise investigate both branches c and ¬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 further the current branche
First 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 \(\Rightarrow\) choose another path
     - if boolean_CSP has a solution \(\Rightarrow\) investigate further the current branche

Boolean abstraction

- hashtable of decisions: keys are decisions, values are Boolean variables
- sub-expressions are shared \(\rightarrow\) rewriting
First prototype
On the fly validation: loops

Let $c$ be the entrance condition and $\text{max}$ the loop bound.

Unfold loop $\text{max}$ times:

1. if $c$ is **trivially simplified** to "true" and $\text{max}$ is **not reached**
   $\iff$ enter the loop

2. if $c$ is **trivially simplified** to "false"
   or if $\text{max}$ is **reached**
   or if $\{ c + \text{linear\_CSP} \}$ is **inconsistent**
   $\iff$ add $\neg c$ to the CSPs and **exit** the loop
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

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

### Table: Experimental Results for an Incorrect Binary Search

<table>
<thead>
<tr>
<th>length</th>
<th>CPBPV (ilog)</th>
<th>ESC/Java</th>
<th>CBMC</th>
<th>WHY inv</th>
<th>BLAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.027s</td>
<td>1.21 s</td>
<td>1.38s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>16</td>
<td>0.037s</td>
<td>1.347 s</td>
<td>1.69s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>32</td>
<td>0.064s</td>
<td>1.792 s</td>
<td>7.62s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>64</td>
<td>0.115s</td>
<td>1.886 s</td>
<td>27.05s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>128</td>
<td>0.241s</td>
<td>1.964 s</td>
<td>189.20s</td>
<td>KO</td>
<td>KO</td>
</tr>
</tbody>
</table>
Binary search

<table>
<thead>
<tr>
<th></th>
<th>length</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPBPV (ilog)</td>
<td>time</td>
<td>1.08s</td>
<td>1.69s</td>
<td>4.04s</td>
<td>17.01s</td>
<td>136.80s</td>
</tr>
<tr>
<td>CBMC</td>
<td>time</td>
<td>1.37s</td>
<td>1.43s</td>
<td>KO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Why</td>
<td>inv</td>
<td></td>
<td></td>
<td>11.18s</td>
<td></td>
<td>KO</td>
</tr>
<tr>
<td>ESC/Java</td>
<td></td>
<td></td>
<td></td>
<td>Error</td>
<td></td>
<td>KO</td>
</tr>
<tr>
<td>BLAST</td>
<td></td>
<td></td>
<td></td>
<td>KO</td>
<td></td>
<td>KO</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>length</th>
<th>CPBPV(ilog)</th>
<th>ESC/Java</th>
<th>CBMC</th>
<th>WHY inv</th>
<th>BLAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.027s</td>
<td>1.21 s</td>
<td>1.38s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>16</td>
<td>0.037s</td>
<td>1.347 s</td>
<td>1.69s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>32</td>
<td>0.064s</td>
<td>1.792 s</td>
<td>7.62s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>64</td>
<td>0.115s</td>
<td>1.886 s</td>
<td>27.05s</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>128</td>
<td>0.241s</td>
<td>1.964 s</td>
<td>189.20s</td>
<td>KO</td>
<td>KO</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>CPBPV(ilog)</th>
<th>ESC/Java</th>
<th>CBMC</th>
<th>Why</th>
<th>BLAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>0.287s</td>
<td>1.828s</td>
<td>0.82s</td>
<td>8.85s</td>
<td>KO</td>
</tr>
</tbody>
</table>
Sum of squares

```c
/*@ 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 */

1 int sum(int[] t, int n)
2     int s = 0;
3     int i = 0;
4     while (i!=t.length)
5         s=s+t[i]*t[i]
6         i =i+1;
7     return s;
```

- Using global constraint `alldiff`
- Solving `non linear` problems
- 66.179s for `n = 10`
On going Work

► New prototype
  ● Written in COMET by Le Vinh Nguyen
  ● NORMAL_IF : turning the program in Conditional Normal Form (Armando et al)
  ● Search driven by the variables in the postconditions

► First Experiment: Flasher manager
  ● Real application provided by GEENSYS (embedded systems development for industrial applications)
  ● Specification given in Simulink
  ● Program: several thousand lines of generated C code
  ● Cannot be handled with first prototype
Flasher manager specification (1)

![Flasher manager diagram](image-url)
Flasher manager specification (2)
Flasher manager: properties

1. Warning has priority over the other functions of the flasher
2. When warning is desactivated left/right flashers remain active if they were active before
3. When the key is turn off, warning and flasher functions are desactivated but not lock/unlock doors functions
4. ...
5. Flashers do not remain continuously activated
Flashe manager: experiments

Violation of property 5

<table>
<thead>
<tr>
<th>N</th>
<th>CBMC</th>
<th>CPBPV COMET</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.003s</td>
<td>0.087s</td>
</tr>
<tr>
<td>30</td>
<td>0.947s</td>
<td>0.901s</td>
</tr>
<tr>
<td>50</td>
<td>2.469s</td>
<td>2.372s</td>
</tr>
<tr>
<td>75</td>
<td>10.634s</td>
<td>2.909s</td>
</tr>
<tr>
<td>100</td>
<td>24.952s</td>
<td>4.751s</td>
</tr>
<tr>
<td>150</td>
<td>52.462s</td>
<td>10.161s</td>
</tr>
</tbody>
</table>
Conclusion

- A novel constraint-programming framework for bounded program verification

- Experimental results
  - Orders of magnitude improvements (running times often independent of the variable domains)
  - Detection of subtle errors in some programs (not found by frameworks based on model checking)

- Further work: improving and generalizing the framework and implementation
  - Optimization of the array implementation
  - Integration of Java objects and references
  - Check for variable overflows
Role of the different solvers

- **LP and 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