CP also meets Software Testing

Arnaud Gotlieb

Certus Software V&V Centre
SIMULA RESEARCH LABORATORY
Lysaker, Norway

CP meets CAV Workshop, Turunc, Turkey
A day in June 2012
CERTUS is also a Centre for research-based innovation (SFI)

Host
Simula Research Laboratory

User partners
CISCO Systems Norway
ESITO
FMC Technologies
KONGSBERG Maritime
TOLL customs and excises

Budget
~10 MNOK (1.3 MEUR) per year over a 8-years period

Origin (2011)
Prof. Lionel Briand (now in Luxembourg)
Industry-driven research problems in Software Validation & Verification

- Certification and verification of real-time embedded software-systems

- Modelling and testing of highly-configurable software-systems

- Automated testing of data-intensive administrative software-systems

With an increasing usage of Constraint Programming techniques (Finite Domains constraint solving, constraint optimization, MIP, Modelling)
A. Time-aware test configurations generation with Constraint Programming

B. Testing deadline misses for real-time systems using constraint-based scheduling techniques

C. Extraction of a formally verified constraint solver for the certification of tax computation
Outline

Constraint-based testing (CBT)

Constraint-based program exploration for automatic test data generation

Constraints over Memory Model Variables for testing pointer programs

Conclusions
Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques (LP, CP, SAT, SMT, ...)

Introduced 20 years ago by Offut and DeMillo in (Constraint-based automatic test data generation IEEE TSE 1991)

Developed in the context of code-based testing and model-based testing

Lots of Research works and tools!
CBT: main tools

CEA - List (Osmose S. Bardin P.Herrmann)
Univ. of Madrid (PET M. Gomez-Zamalloa, E. Albert, G. Puebla)
Univ. of Stanford (EXE D. Engler, C. Cadar, P. Guo)
Univ. of Nice Sophia-Antipolis (CPBPV M. Rueher, H. Collavizza, P.V. Hentenryck)
INRIA - Celtique (Euclide, JAUT A. Gotlieb, F. Charreteur)

Tools with external industrial usage:

GATEL (CEA B. Marre, since 2004)
Test Designer (Smartesting B. Legeard, since 2003)
PEX (Microsoft P. de Halleux, N. Tillmann, since 2009)

Tools with internal industrial usage:

Inka V1 (Dassault A. Gotlieb, B. Botella, in 2001)
PathCrawler (CEA N. Williams, since 2004)
SAGE (Microsoft P. Godefroid, since 2010)
The automatic test data generation problem

Given a location k in a program under test, generate a test input that reaches k

Reachability problem in infinite-state systems is undecidable in general!

Even when adding bounds, hard combinatorial problem

Using Random Testing,

$$\text{Prob}\{\text{reach } k\} = 2 \over 2^{32} \times 2^{32} \times 2^{32} = 2^{-95} = 0.00000\ldots1.$$ 

Constraint solving techniques are required!

- Loops (i.e., infinite-state systems) and infeasible paths
- Pointers, dynamic structures, higher-order computations (virtual calls)
- Floating-point computations, modular computations
Context of this talk

**Code-based testing**

Imperative programs \((C, \ldots)\)

Programs with loops

Single-threaded programs

Selected location in code

(not model-based testing)

(not Functionnal P., not Logic P., not Object-Oriented P.)

(i.e., infinite-state systems)

(no concurrent or parallel programs)

(i.e., reachability problems)
Constraint-based program exploration for automatic test data generation
A reachability problem

\[
f(\text{int } i, \ldots )
\{
\text{a. } j = 100; \\
\text{while( } i > 1) \\
\text{b. } \{ j++ ; i-- ; \}
\]

\[
\ldots
\]

d. if( j > 500)

e. \ldots

value of i to reach e?
Path-oriented exploration

```c
f(  int i, ...  )
{
a.    j = 100;
    while( i > 1)
    { j++ ; i-- ;}
b.        ;

d.  if( j > 500)
e.    ...
```

1. Path selection
e.g., (a-b)^14-...-d-e

2. Path condition generation (via symbolic exec.)
j_1=100, i_1>1, j_2=j_1+1, i_2=i_1-1, i_2>1,..., j_{15}>500

3. Path condition solving
   unsatisfiable → FAIL

Even without loops, #paths is exponential with #decisions

Backtrack!
f( int i, ... )
{
a.   j = 100;
    while( i > 1)
    { j++ ; i-- ;}
   ...  

1. Constraint model generation

2. Control dependencies generation;
   \( j_1=100, i_3 \leq 1, j_3 > 500 \)

3. Constraint model solving
   \( j_1 \neq j_3 \) entailed \( \Rightarrow \) unroll the loop 400 times \( \Rightarrow i_1 \) in \( 401 .. 2^{31}-1 \)

No backtrack!
Constraint-based program exploration

- Based on a constraint model of the whole program (i.e., each statement is seen as a relation)
- Constraint reasoning over control structures
- Requires to build dedicated constraint solvers:
  * propagation queue management with priorities
  * specific propagators and meta-constraints
  * structure-aware labelling heuristics (Systematic search over finite domains)

Prototype tools:  **Inka** (Gotlieb Botella Rueher ISSTA’98)  
**Euclide** (Gotlieb ICST’09)
Assignment as Constraint

Viewing an assignment as a relation requires to normalize expressions and rename variables (through single assignment languages, e.g. SSA)

\[ \text{i}^* = ++\text{i}; \quad \rightarrow \quad \text{i}_2 = (\text{i}_1+1)^2 \]

Using bound-consistency filtering over finite domains:

- \( \text{i}_1 = 3 \) ?
- \( \text{i}_1 \) in -4..2
- no
- \( \text{i}_1 \) in -5..3

- \( \text{i}_2 = 16 \)
- \( \text{i}_2 = 9 \) ?
- \( \text{i}_2 = 7 \) ?
- \( \text{i}_2 \) in 5..16 ?
Statements as constraints

✓ Type declaration:  
\[
\text{signed long } x; \quad \rightarrow \quad x \text{ in } -2^{31}..2^{31}-1
\]

✓ Assignments:  
\[
i^* = ++i; \quad \rightarrow \quad i_2 = (i_1+1)^2
\]

✓ Memory and array accesses and updates:  
\[
v = A[i] \quad \text{(or } p = \text{Mem}[&p]) \quad \rightarrow \quad \text{variations of element/3}
\]

✓ Control structures: dedicated meta-constraints 
(interface, awakening conditions and filtering algorithms)

\[
\text{Conditionnals (SSA)} \quad \text{if } D \text{ then } C_1, \text{ else } C_2 \quad \rightarrow \quad \text{ite/6}
\]

\[
\text{Loops (SSA)} \quad \text{while } D \text{ do } C \quad \rightarrow \quad \text{w/5}
\]
Conditional as meta-constraint: ite/6

\[
\text{ite}(x > 0, j_1, j_2, j_3, \ j_1 = 5, \ j_2 = 18) \ \text{iff}
\]

- \(x > 0\) \(\rightarrow\) \(j_1 = 5 \land j_3 = j_1\)
- \(\neg(x > 0)\) \(\rightarrow\) \(j_2 = 18 \land j_3 = j_2\)
- \(\neg(x > 0 \land j_1 = 5 \land j_3 = j_1)\) \(\rightarrow\) \(\neg(x > 0) \land j_2 = 18 \land j_3 = j_2\)
- \(\neg(\neg(x > 0) \land j_3 = j_2)\) \(\rightarrow\) \(x > 0 \land j_1 = 5 \land j_3 = j_1\)
- Join(\(x > 0 \land j_1 = 5 \land j_3 = j_1, \ \neg(x > 0) \land j_1 = 18 \land j_3 = j_2)\)

Implemented as a new global constraint
(interface, awakening conditions, filtering algo.)
Loop as meta-constraint: w/5

\[ v_3 = \phi (v_1, v_2) \]
while (Dec)

- Dec\(_{V_3 \leftarrow V_1} \rightarrow \) body\(_{V_3 \leftarrow V_1} \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2 \leftarrow V_{\text{new}}})
- \neg\text{Dec}_{V_3 \leftarrow V_1} \rightarrow v_3 = v_1

- \neg(\text{Dec}_{V_3 \leftarrow V_1} \land \text{body}_{V_3 \leftarrow V_1}) \rightarrow \neg\text{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1
- \neg(\neg\text{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1) \rightarrow \text{Dec}_{V_3 \leftarrow V_1} \land \text{body}_{V_3 \leftarrow V_1} \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2 \leftarrow V_{\text{new}}})
- \text{join}(\text{Dec}_{V_3 \leftarrow V_1} \land \text{body}_{V_3 \leftarrow V_1} \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2 \leftarrow V_{\text{new}}}), \neg\text{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1)
f( int i ) {
    j = 100;
    while ( i > 1 )
    {
        j++ ; i-- ;
    }

    ...  

    if ( j > 500 )
    
    ...  

    w( Dec, V_1, V_2, V_3, body ) :-
    • Dec_{V_3 \leftarrow V_1} \rightarrow body_{V_3 \leftarrow V_1} \land w( Dec, V_2, v_{new}, V_3, body_{V_2 \leftarrow v_{new}} )
    • \neg Dec_{V_3 \leftarrow V_1} \rightarrow v_3 = v_1 
    • \neg ( Dec_{V_3 \leftarrow V_1} \land body_{V_3 \leftarrow V_1} ) \rightarrow \neg Dec_{V_3 \leftarrow V_1} \land v_3 = v_1 
    • \neg ( \neg Dec_{V_3 \leftarrow V_1} \land v_3 = v_1 ) \rightarrow Dec_{V_3 \leftarrow V_1} \land body_{V_3 \leftarrow V_1} \land w( Dec, V_2, v_{new}, V_3, body_{V_2 \leftarrow v_{new}} )
    • join( Dec_{V_3 \leftarrow V_1} \land body_{V_3 \leftarrow V_1} \land w( Dec, V_2, v_{new}, V_3, body_{V_2 \leftarrow v_{new}} , 
        \neg Dec_{V_3 \leftarrow V_1} \land v_3 = v_1 )

    i = 23, j_1 = 100 ?
    no
    i in 401..2^{31}-1

    w( i_3 > 1, (i,j_1), (i_2,j_2), (i_3,j_3), j_2 = j_3 + 1 \land i_2 = i_3 - 1 )

    i_3 = 1, j_3 = 122
    i_3 = 10 ?
    j_1 = 100, j_3 > 500 ?
Features of constraint-based exploration

✓ Special meta-constraints implementation for ite and w

By construction, \( w \) is unfolded only when necessary but \( w \) may NOT terminate!
\( \rightarrow \) only a semi-correct test data generation procedure

✓ Join is implemented using Abstract Interpretation operators (e.g., interval-based union, weak-join operator, widening in Euclide)

✓ Special propagators based on linear-based relaxations Using Linear Programming over rationals (i.e., \( \mathbb{Q} \)-polyhedra)

Abstraction-based relaxations
Abstraction-based relaxations

→ During constraint propagation, constraints can be relaxed in Abstract Domains (e.g., Q-Polyhedra, Octagons, ...)

\[ Z = X \times Y, \quad X \text{ in } a..b, \ Y \text{ in } c..d \]

⇔ \{ Z - Ya - Xc + ac \geq 0, \\
    Xd - Z - ad + aY \geq 0, \\
    bY - bc - Z + Xc \geq 0, \\
    bd - bY - Xd + Z \geq 0, \\
    a \leq X \leq b, \ c \leq Y \leq d \} 

→ To benefit from specialized algorithm (e.g., simplex for linear constraints) and capture global states of the constraint system

→ Require safe/correct over-approximation (to preserve property such as: if the Q-Polyhedra is void then the constraint system is unsatisfiable)

→ Q-Polyhedra in Euclide, implementing Dynamic Linear Relaxation, propagation queue with priorities
Abstraction-based relaxations: weak-join operator
(Sankaranarayanan et al. VMCAI’06)

Join operations can be realized by convex hull, but usually too costly!
In Euclide, we took advantage of the weak-join of Q_ployhedra
(based on simplex calculations)
Abstraction-based relaxations: weak-join operator
(Sankaranarayanan et al. VMCAI’06)
Abstraction-based relaxations: weak-join operator
(Sankaranarayanan et al. VMCAI’06)

Weak_join operator

\[
\text{The disjunction: } \bigvee_{i \in I} \left\{ g_1^i(x) \geq c_1^i \right\}, \bigvee_{i \in I} \left\{ g_2^i(x) \geq c_2^i \right\}
\]
\[
x = (x_1, \ldots, x_n), \text{ where } x_i \in Z
\]

Weak_join:
\[
\alpha_1 = \text{Minimize } g_1^1(x) \text{ subject to } \left\{ g_2^i(x) \right\}_{i \in I}
\]
\[
\vdots
\]
\[
\alpha_p = \text{Minimize } g_1^{\text{card}(I)}(x) \text{ subject to } \left\{ g_2^i(x) \right\}_{i \in I}
\]
\[
\alpha_{p+1} = \text{Minimize } g_1^1(x) \text{ subject to } \left\{ g_1^i(x) \right\}_{i \in I}
\]
\[
\vdots
\]
\[
\alpha_{2p} = \text{Minimize } g_2^{\text{card}(I)}(x) \text{ subject to } \left\{ g_1^i(x) \right\}_{i \in I}
\]
\[
g_1^1(x) \geq \text{Min}(\alpha_1, c_1^1),
\]
\[
\vdots
\]
\[
g_2^{\text{card}(I)}(x) \geq \text{Min}(\alpha_{2p}, c_2^{\text{card}(I)})
\]
Constraint-based program exploration

- Handles loops in constraint-based test data generation, without bounding the number of iterations;

- Useful for reaching a particular uncovered location in the code (complement an existing test set generated by « systematic » path-exploration)

- Use of the global constraint interface in clpfd to implement w, or dedicated solver (propagation queue management)

- May not terminate, timeout needed!

Foundations of the approach (Gotlieb Botella Rueher ISSTA’98, SEN’98, CL’00)
Abstraction-based relaxation (Denmat Gotlieb Ducassé ISSRE’07)
Global constraint w, extended with widening (Denmat Gotlieb Ducassé CP’07)
Euclide: A Constraint-based testing platform for C (Gotlieb ICST’09)
Application on the TCAS case study (Gotlieb KER Journal 2012)
Constraints over Memory Model Variables for testing pointer programs
Constraints over memory models: aliasing problems

How to apply constraint-based reasoning over statement like \( *p := *p + 1 \) ?

- Then fail or exception

- Then \( a_2 = a_1 + 1 \)

- Then \( a_2 = a_1 + 1 \) or \( b_2 = b_1 + 1 \)

- Then \( p_2 = p_1 + 1 \), meaning that \( p \) now refers to the next memory location
Our propositions

How to represent abstract memories and to reason on them?

1) Constraint reasoning over Memory, as a set of graphs (Gotlieb et al., ASE’05, IST 2007)

2) Constraint reasoning over Memory, as a structured set of unbounded arrays (Charreteur et al., JSS 2009)
Weaknesses of our first memory model

- Requires a preliminary points-to analysis that may be too imprecise when dynamic (de-)allocation is involved

- Pointers as function inputs, can point to anything on the heap

- Some conditions may constrain the shape of dynamic data structures. How to handle this in a constraint solver?

```c
int P(struct cell * t) {  
    if( t == t->next ) { …
}
```

Diagram:

```
  t
  |  
  v
next
```

constrains t to
Memory, as a structured set of unbounded arrays

\[ M : \text{memory} \]
- **Integers**: \( \text{TABi} \)
- **Floats**: \( \text{TABf} \)
- **Pointers**: \( \text{TABp} \)
- **Structures**: [\( S_1, S_2, \ldots \)]

\[ S : \text{structure} \]
- **status**: closed or not
- **cont.**: \( \{ @_i \} \)

\[ \text{TAB} : \text{tableau} \]
- **status**: closed or not
- **cont.**: \( \{ @_i - V_i, \ldots \} \)

\[ V : \text{integer within a finite domain} \]
- **Type**: 16, 32, 64 bits, signed, unsigned
- **dom**: \{possible values\}
- **Min..Max**

\[ V : \text{float within an interval} \]
- **Type**: float (32), double (64)
- **dom.**: Min..Max

\[ V : \text{pointer} \]
- **possibly_null**: yes, no
- **dom**: \{possible values\}
- **nondom**: \{non-possible values\}
Introducing constraints on memories

• Memories = unknowns representing states (sets of pairs Address-Value)

• Relations on these unknowns, constraint reasoning on these unknowns

C program

<table>
<thead>
<tr>
<th>C program</th>
<th>Constraints store</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = i + 1 --------&gt;</td>
<td>load_elt(@i, I_1, M_1)</td>
</tr>
<tr>
<td></td>
<td>I_2 = I_1 + 1</td>
</tr>
<tr>
<td></td>
<td>store_elt(@i, I_2, M_1,M_2)</td>
</tr>
<tr>
<td></td>
<td>*p = 3 ---------&gt;</td>
</tr>
<tr>
<td></td>
<td>load_elt(@p, P_1, M_2)</td>
</tr>
<tr>
<td></td>
<td>DP_1 = 3</td>
</tr>
<tr>
<td></td>
<td>store_elt(P_1,DP_1,M_2,M_3)</td>
</tr>
<tr>
<td></td>
<td>j = i + 2 ---------&gt;</td>
</tr>
<tr>
<td></td>
<td>load_elt(@i,I_3,M_3)</td>
</tr>
<tr>
<td></td>
<td>J_1 = I_3 + 2</td>
</tr>
<tr>
<td></td>
<td>store_elt(@j,J_1, M_3,M_4)</td>
</tr>
</tbody>
</table>
Constraints on memories

- `new_elt(TYPE, X, V_INIT, M0, M1, ENV)`
- `delete_elt(TYPE, X, M0, M1, ENV)`
- `load_elt(TYPE, X, VALUE, M, ENV)`
- `store_elt(TYPE, X, VALUE, M0, M1, ENV)`

- `M1 = M2 /* Useful in control structures */`
  - `closed(M)`
  /* Useful to closed the memory during final search */
store_elt(P, V, M1, M2)

M1:
Status: not closed
Includes:
i – Vi
j – Vj
k – Vk
...

P:
Domain pointer
{i,j}

V:
Domain Integer
1..5

M2:
Status: not closed
Includes:
i – Vi’
j – Vj’
k – Vk’...

Store_elt

store_elt(P, V, M1, M2)
store_elt(P, V, M1, M2)

M1:
Status: not closed
Includes:
i – Vi → 1.. 2
j – Vj → 5.. 9
k – Vk → 2
...

M2:
Status: not closed
Includes:
i – Vi’ → 3.. 6
j – Vj’ → 7..18
k – Vk’ → ?
...

P:
Domain pointer
{i, j}

V:
Domain Integer
1.. 5

Automatic deductions after the constraint propagation step:
P = i, V = Vi’ in 3..5, Vj = Vj’ in 7..9, Vk = Vk’ = 2
Model for the definition of a new constraint

\[ S_{VAR} \]

- Constraints
- Store
- Awake
- Suspend
- Reduce
- Success
- Fail
- Exit

\[ S_{VAR} \]
store_elt(P, V, M1, M2)
Conclusions
What was left apart in my talk

- **Constraints over floating-point variables**: FPSE Solver  
  (Botella Gotlieb Michel STVR 2006, Carlier Gotlieb ICTAI’11)

- **Constraints over modular integers** (Gotlieb Leconte Marre ModRef’10)

- **Constraints over memory models for Java Bytecode (i.e., with inheritance and virtual method calls)** (Charreteur Gotlieb ISSRE’10)

- **Uniform random generation** of test data in path testing  
  (Gotlieb Petit CP’07, JSS’10)

- **Explanation-based generalization of infeasible paths** in  
  *Dynamic Symbolic Execution*  
  (Delahaye Botella Gotlieb ICST’10, TSE in rev)
• Applications to the testing of critical embedded software
  - BCE ABE Rafale (2001)
  - Java Card (2004-2005)
  - TCAS SIR (2008)
  - TCAS unmanned planes (2011)

• Development of 4 Research prototype tools:
  Inka, Euclide, PRT and FPSE
  (more than 45KLOC Prolog, Java, C, Tcl/Tk)

• Research projects: INKA, DANOCOPS, CASTLES, ACI V3F, ANR CAT/U3CAT, ANR CAVERN...
Conclusions

• Emerging concept in code- and model-based software testing

• Constraint Programming techniques offers:
  - Global constraint design
  - disjunctive constraint programs in a constructive way.
  - Time-aware optimization through branch&bound is given for free
  - but unsatisfiability detection has to be improved (e.g., by combining techniques SMT/CP)

• Mature tools (academic and industrial) already exist, but application on real-sized industrial cases still have to be demonstrated
Further work

- Array constraint solving. (More global reasoning are required!)

  A combined SMT/CP approach for solving constraints with arrays and arithmetics. Constraint solver CCFD and large experimental validation over random formulas.

  joint work with S. Bardin from CEA

- Improving constraint-reasoning over function calls, modelling function calls as global constraints

- Dedicated labelling search, exploiting the structure of the programme
Thank you!

- **PhD students**
  Tristan Denmat, Matthieu Petit, Florence Charreteur, Mickael Delahaye, Nadjib Lazaar, Aymeric Hervieu

- **Post-doc**
  Sandrine Gouraud, Pierre Rousseau, Matthieu Carlier

- **Co-authors**
  Olivier Lhomme, Michel Rueher, Claude Michel, Yahia Lebbah, Michel Leconte, Mireille Ducassé, Bernard Botella, Patrick Taillibert, Franck Calvet, Bruno Marre, Benjamin Blanc, Frédéric Dadeau, Nicky Williams, Catherine Dubois, Patrick Bernard, Matthieu Wattel, Benoît Baudry, Sébastien Bardin, Lionel Briand