An overview of

Constraint-Based Testing

Arnaud Gotlieb
SIMULA RESEARCH LAB.

Certus CRIM, Montréal, Nov. 2015 sfi Centre for Research-based Innovation

Certus sfi Centre for Research-based Innovation

The Certus Centre

Hosted by SIMULA

Established and awarded SFI in Oct. 2011

duration: 8 years
RCN funding: ~10MEUR

www.certus-sfi.no


Cisco Systems Norway


Norwegian Custom and excise


ABB Robotics Stavanger

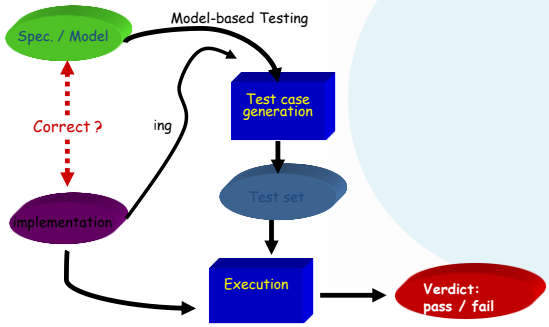

Kongsberg Maritime

esito

CANCER Registry of Norway

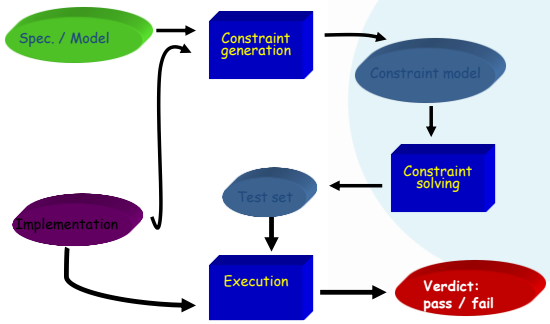
[simula . research laboratory]
- by thinking constantly about it

Software Testing



Certus sfi Centre for Research-based Innovation

Constraint-Based Testing



Certus sfi Centre for Research-based Innovation

Constraint-Based Testing (CBT)

Constraint-Based Testing (CBT) is the process of **generating test cases** against a **testing objective** by using **constraint solving techniques**

Introduced 25 years ago by Offut and DeMillo in
(Constraint-based automatic test data generation IEEE TSE 1991)

Success stories in the context of code-based testing with code coverage objectives (Microsoft, Thales, CEA), and Model-Based Testing (Smartesting)

Lots of Research works and tools !

Certus sfi Centre for Research-based Innovation

The automatic test data generation problem

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

Undecidable in general, but ad-hoc methods exist

- ✓ Highly combinatorial

```
f (int x1, int x2, int x3) {
    if(x1 == x2 && x2 ==x3)
      if(x3==x1*x2) ...
                          }
```

Here, with random testing, Prob{ reack k} = 2 over $2^{32} \times 2^{32} \times 2^{32} = 2^{-95} = 0.00000\ldots1$

- ✓ Loops and non-feasible paths
- ✓ Modular integer and floating-point computations
- ✓ Pointers, dynamic structures, function calls, …

Context of this overview

Code-based testing	(not model-based testing)
Imperative programs (C, ADA, …)	(neither Functionnal P., nor Logic P.)
Programs with loops and recursion	(i.e., infinite-state systems)
Single-threaded programs	(no concurrent or parallel programs)
Selected location in code	(i.e., reachability problems)

Outline

- Introduction
- Path-oriented exploration
- Constraint-based exploration
- Further work

Certus

sfi = Centre for Research-based Innovation

Path-oriented test data generation

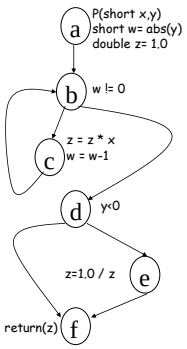
- Select one or several paths → **Path selection** step
- Generate the path conditions → **Symbolic evaluation** techniques
- Solve the path conditions to generate test data that activate the selected paths → **Constraint solving**

Test objectives:
generating a test suite that covers a given testing criterion
(all-statements, all-decisions, all-paths…)
or a test data that raise a safety or security problem
(assertion violation, buffer overflow, …)

Main CBT tools: **ATGen** (Meudec 2001), **EXE** (Cadar et al. 2006), **ECLAIR** (Bagnara Bagnara Gori 2013)

Path selection on an example

```
double P(short x, short y) {
    short  w = abs(y) ;
    double z  = 1.0 ;
    while ( w != 0 )
    {
      z  = z * x ;
      w = w - 1 ;
    }
    if ( y<0 )
     z = 1.0 / z ;
   return(z) ;
   }
```



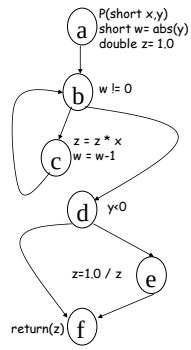
Path selection on an example

all-statement coverage:
a-b-c-b-d-e-f

All-decisions coverage:
a-b-c-b-d-e-f
a-b-d-f

all-2-paths (at most 2 times in loops):
a-b-d-f
~~a-b-d-e-f~~
…
a-b-(c-b)²-d-e-f

all-paths:
Impossible



Path condition generation

Symbolic state: <Path, State, Path Conditions>

Path = n_i-..-n_j is a path expression of the CFG
State = $\langle v_i, \varphi_i \rangle_{v \in Var(P)}$ where φ_i is an algebraic expression over **x**
Path Cond. = $c_1,..,c_n$ where c_i is a condition over **x**

x denotes symbolic variables associated to the program inputs and `Var(P)` denotes internal variables

Symbolic execution

Ex : a-b-(c-b)²-d-f with X,Y

<a, <z,1>, <w,abs(Y)>, true >

<a-b, <z,1>, <w,abs(Y)>, abs(Y) != 0 >

<a-b-c, <z,X>, <w,abs(Y)-1>, abs(Y) != 0 >

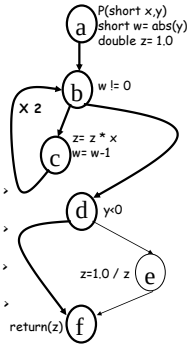
<a-b-c-b, <z,X>, <w,abs(Y)-1>, abs(Y) != 0, abs(Y)-1 != 0 >

< a-b-c-b-c, <z,X²>, <w,abs(Y)-2>, abs(Y) != 0, abs(Y)-1 != 0 >

<a-b-(c-b)², <z,X²>, <w,abs(Y)-2>, abs(Y) != 0, abs(Y) != 1, abs(Y)-2 = 0 >

<a-b-(c-b)²-d, <z,X²>, <w,abs(Y)-2>, abs(Y) != 0, abs(Y) != 1, abs(Y) = 2, Y ≥ 0 >

<a-b-(c-b)²-d-f, <z,X²>, <w,0>, Y=2 >



Computing symbolic states

- `<Path, State, PC>` is computed by induction over each statement of Path
- When the Path conditions are unsatisfiable then `Path` is non-feasible and reciprocally (i.e., symbolic execution captures the concrete semantics)

  ex : ~~<a-b-c-c-b-c-f, {…}, abs(Y)=0 ∧ Y<0 >~~

- Forward vs backward analysis:

  Forward → interesting when states are needed
  Backward → saves memory space, as complete states are not computed

Backward analysis

Ex : a-b-(c-b)²-d-f with X,Y

f,d: **Y ≥0**

b: **Y ≥0**, w = 0

c: **Y ≥0**, w-1 = 0

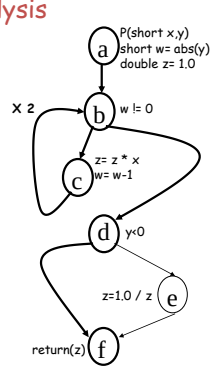
b: **Y ≥0**, w-1 = 0, w != 0

c: **Y ≥0**, w-2 = 0, w-1 != 0

b: **Y ≥0**, w-2 =0, w-1 != 0,w != 0

a: **Y ≥0, abs(Y)-2 = 0, abs(Y)-1 != 0, abs(Y) != 0**

Y = 2



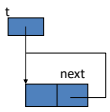
Problems for symbolic evaluation techniques

→ Combinatorial explosion of paths

→ Symbolic execution constrains the shape of dynamically allocated objects

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

constrains t to:



(Modelling dynamic memory management in constraint-based testing.
Charreteur Botella Gotlieb JSS 09)
(Constraint-based test input generation for java bytecode.
Charreteur Gotlieb ISSRE 10)

→ Floating-point computations

```
float foo( float x) {
   float y = 1.0e12, z ;
1. if( x < 10000.0 )
2.     z = x + y;
3. if( z > y)
4.     …
```

Is the path 1-2-3-4 feasible ?

Path conditions:

x < 10000.0

x + 1.0e12 > 1.0e12

On the reals : x ∈ (0,10000)

On the floats : no solution !

Conversely,

```
float foo( float x) {
   float y = 1.0e12, z ;
1. if( x > 0.0 )
2.     z = x + y;
3. if( z == y)
4.     …
```

Is the path 1-2-3-4 feasible ?

Path conditions:
x > 0.0
x + 1.0e12 = 1.0e12

→ On the reals : no solution
→ On the floats: x ∈ (0, 32767.99…)

Solution: build a dedicated constraint solver over the floats !

(Symbolic execution of floating-point computations,
Botella Gotlieb Michel, STVR 2006
Bagnara Carlier Gotlieb Gori, ICST 2013, JoC 2015)

Dynamic symbolic evaluation

- Symbolic execution of a concrete execution (also called concolic execution)
- By using input values, **feasible paths only** are (automatically) selected
- Randomized algorithm, implemented by instrumenting each statement of P

Main CBT tools:

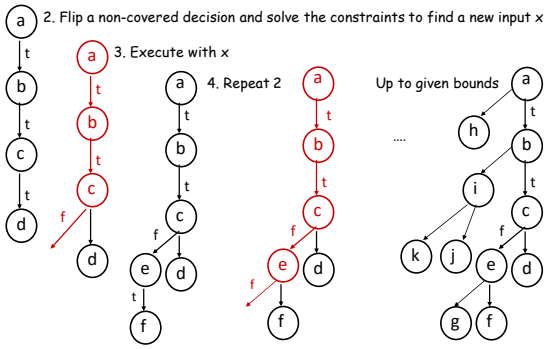
PathCrawler (Williams et al. 2005),
PEX (Tillman et al. Microsoft 2008),
SAGE (Godefroid et al.2008)
KLEE (Cadar et al. 2008)

Comes in two ingredients…

1st ingredient: path exploration

1. Draw an input at random, execute it and record path conditions
2. Flip a non-covered decision and solve the constraints to find a new input x
3. Execute with x
4. Repeat 2

Up to given bounds



2nd ingredient: use concrete values

- Use actual values to simplify the constraint set

Flip If($x_3 == x_1 * x_2$) … ($x_1 = 6, x_2=7$)

(1) Exact solving -- add $x_3 != x_1 * x_2$ to the constraint solver

(2) Approximate solving -- add $x_3 != 6 * x_2$ && $x_1=6$

or -- add $x_3 != x_1 * 7$ && $x_2=7$

(3) Useless solving -- add $x_3 != 42$ && $x_1=6$ && $x_2=7$

PathCrawler: (1) **PEX**: (2) **SAGE**: (3) and then (2)

Constraint solving in symbolic evaluation

- Mixed Integer Linear Programming approaches
  (i.e., simplex + Fourier's elimination + branch-and-bound)
  CLP(R,Q) in **ATGen** (Meudec 2001)
  Ipsolve in **DART/CUTE** (Godefroid/Sen et al. 2005)
- SMT-solving (= SAT + Theories)
  STP in **EXE and KLEE** (Cadar et al. 2006)
  Z3 in **PEX and SAGE** (Tillmann and de Halleux 2008)
- Constraint Programming techniques (constraint propagation and labelling)
  Colibri in **PathCrawler** (Williams et al. 2005)
  Disolver in **SAGE** (Godefroid et al. 2008)
  ECLAIR (Bagnara Bagnara Gori 2013)

Outline

- Introduction
- → Path-oriented exploration
- Constraint-based exploration
- Further work

Certus

sfi Centre for Research-based Innovation

Constraint-based program exploration

- Based on a constraint model of the whole program
  (i.e., each statement is seen as a relation between two memory states)

- Constraint reasoning over control structures

- Requires to build **dedicated constraint solvers**:
  * propagation queue management with priorities
  * specific propagators and global constraints
  * structure-aware labelling heuristics

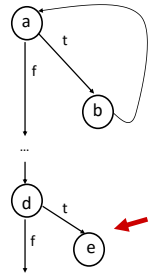
Main CBT tools: **InKa** (Gotlieb Botella Rueher 1998),
GATEL (Marre 2004),
Euclide (Gotlieb 2009)

A reacheability problem

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

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

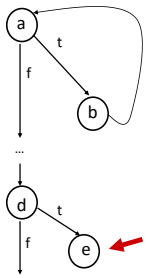
value of i to reach e ?



Path-oriented exploration

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

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



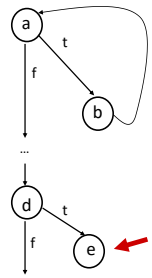
1. Path selection
   e.g., $(a\text{-}b)^{14}$-...-d-e
2. Path conditions generation (via symbolic exec.)
   j_1=100, i_1>1, j_2=j_1+1, i_2=i_1-1, i_2>1,..., j_{15}>500
3. Path conditions solving
   unsatisfiable → FAIL

Backtrack !

Constraint-based exploration

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

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



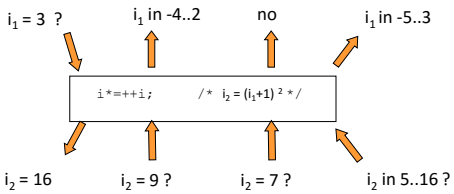
1. Constraint model generation (through SSA)
2. Control dependencies generation;
   j_1=100, $i_3 \leq 1$, $j_3 > 500$
3. Constraint model solving
   $j_1 \neq j_3$ entailed → unroll the loop 400 times → i_1 in 401 .. 2^{31}-1

No backtrack !

Assignment as Constraint

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

`i*=++i ;` ⟶ $i_2 = (i_1+1)^2$



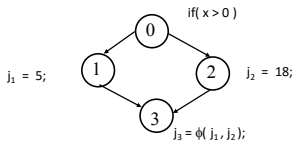
Statements as (global) constraints

- ✓ Type declaration: `signed long x;` → x in $-2^{31}..2^{31}-1$
- ✓ Assignments: `i*=++i ;` → $i_2 = (i_1+1)^2$
- ✓ Control structures: dedicated global constraints

  Conditionnals (SSA) `if D then` C_1 `; else` C_2 `;` $v_3=\phi(v_1,v_2)$ → ite/6

  Loops (SSA) $v_3=\phi(v_1,v_2)$ `while D do C` → w/5

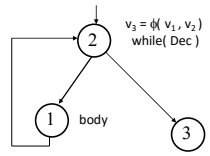
Conditional as global constraint: ite/6



ite(x > 0, j_1, j_2, j_3, $j_1 = 5$, $j_2 = 18$) iff

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

Loop as global constraint: w/5



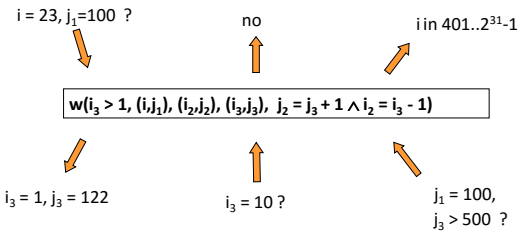
w(Dec, V_1, V_2, V_3, body) iff

- $Dec_{V3 \leftarrow V1} \rightarrow body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $\neg Dec_{V3 \leftarrow V1} \rightarrow v_3 = v_1$
- $\neg(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1}) \rightarrow \neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1$
- $\neg(\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1) \rightarrow Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $join(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$, $\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1)$

```
f(  int i  ) {
  j = 100;
  while( i > 1)
   { j++ ; i-- ;}
  ...
  if( j > 500)
    ...
```

w(Dec, V_1, V_2, V_3, body) :-

- $Dec_{V3 \leftarrow V1} \rightarrow body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $\neg Dec_{V3 \leftarrow V1} \rightarrow v_3 = v_1$
- $\neg(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1}) \rightarrow \neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1$
- $\neg(\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1) \rightarrow Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $join(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$, $\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1)$



Features of the w relation

- ✓ It can be nested into other relation (e.g., nested loops w($cond_1$, v_1,v_2,v_3, w($cond_2$, ...))
- ✓ Managed by the solver as any other constraint (its consistency is iteratively checked, awakening conditions, success/failure/suspension)
- ✓ By construction, w is unfolded only when necessary but w may NOT terminate !
- ✓ Join is implemented using Abstract Interpretation operators (interval union, weak-join, widening)

(Gotlieb et al. CL'2000, Denmat Gotlieb Ducassé ISSRE'07 and CP'2007)

Outline

- Introduction
- Path-oriented exploration
- Constraint-based exploration
- Further work

CBT (summary)

- Emerging concept in code-based automatic test data generation
- Two main approaches:

  Path-oriented test data generation ***vs*** constraint-based exploration

- Constraint solving:

  - Linear programming
  - SMT-solvers
  - Constraint Programming techniques with *abstraction-based relaxations*

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

Certus

sfi Centre for Research-based Innovation

CBT: Pros/Cons

Pros: Handle control and data structures (i.e., pure SAT-solving doesn't work well in that context !) in an efficient way

100%-coverage of testing criteria as required by standards for critical software (e.g., DO-178, Misra, ISO 2626.2)

Fully automated test data generation methods

Cons:

No semantics description, no formal proof → correction is not a priority !

Unsatisfiability detection has to be improved (to avoid costly labelling), by combining techniques (e.g., SMT/CP)

Exploration techniques do not currently keep track of previous solved constraint systems

Further work

- Combining SMT-solving and /CP-based solver (PhD Q. Plazar, joint work with CEA, France)
- Constraint optimization models for test suite execution scheduling
- Constraint-Based Testing from **feature models**, in the context of **Software Product Lines Testing**

Thanks !

Certus

sfi Centre for Research-based Innovation