Code-based Testing with Constraints

Arnaud Gotlieb
SIMULA RESEARCH LAB.

HUAWEI, Paris, 31 Mar. 2021
The VIAS Dept. at Simula Research Laboratory

2 Permanent scientists, 3 PhD Students, 2 Postdoc  [+ 2 Postdoc, 1 PhD in 2021]

VIAS: Validation Intelligence of Autonomous Software Systems

Organized as a Research-Based Innovation Center until Dec. 2019

Strongly involved into the AI4EU Project (H2020, 2019-2021)
European AI-on-demand platform
and the TRANSACT Project (ECSEL, 2021-2024)

Created RESIST, the first Inria-Simula associate team in 2021.
Outline

Software Testing and Code-based Testing

Path-oriented exploration

Constraint-based exploration

Summary and further work
Software Testing

User Requirements

Informal World

Mathematical World

Model-based Testing

Constraint Model

Conformity?

Spec / Model

Oracle

Verdict: Pass or Fail

Logical World

Conformity?

Source Code

Conformity?

Physical World

Test inputs

Conformity?

Executable Under Test

Test results

Exigences

User Requirements

User Requirements

Source Code

Spec / Model
Code-based testing aims at generating test inputs such that selected code locations are executed.

**Test inputs generation is a cognitively complex task:**
- Requires to “understand” the code in order to find test inputs
- Program’s input space is usually very large (sometimes unbounded)
- Complex software code (e.g., solving ODEs or PDEs) are difficult to test
- Code optimizations can often only be tested with code-based testing

**Not easily amenable to automation:**
- Automatic test inputs generation is undecideable in the general case!
- Exploring the input space yields to combinatorial explosion
- Control and data structures requires dedicated treatments
The automatic test input 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

\[
f (\text{int } x_1, \text{int } x_2, \text{int } x_3) \{
    \text{if}(x_1 == x_2 \&\& x_2 == x_3) \\
    \text{if}(x_3 == x_1 * x_2) ...
\}
\]

location k

Here, with random testing, \( \text{Prob}\{\text{reach } k\} = \frac{2}{2^{32} \times 2^{32} \times 2^{32}} = 2^{-95} = 0.00000...1 \)

So, constraint solving is crucial to address this problem in an efficient way

✓ Loops and non-feasible paths
✓ Modular integer and floating-point computations
✓ Pointers, dynamic structures, function calls, …
✓ Inheritance, polymorphism
Constraint-Based Testing (CBT) is the process of **generating test cases** against a **testing objective** by using **constraint solving techniques**.

Introduced 30 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, CEA, Smartesting, Conformiq, Thales, …)

Lots of Research works and tools!
Outline

Software Testing and Code-based Testing

Path-oriented exploration

Constraint-based exploration

Summary and further work
Path-oriented test data generation

Select one or several paths → 1. Path selection

Generate the path conditions → 2. Symbolic evaluation

Solve the path conditions to generate test data that activate the selected paths → 3. 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 2006)
**ECLAIR** (Bagnara 2013), **BINSEC** (Bardin 2015, 2020)
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

```c
P(short x,y)
short w= abs(y)
double z= 1.0
return(z)
```
Symbolic state: $<\text{Path, State, Path Conditions}>$

Path $= n_i \ldots n_j$ is a path expression of the CFG

State $= <v_i, \varphi_i>_{v \in \text{Var}(P)}$ where $\varphi_i$ is an algebraic expression over $X$

Path Cond. $= c_1, \ldots, c_n$ where $c_i$ is a condition over $X$

$X$ denotes symbolic variables associated to the program inputs and $\text{Var}(P)$ denotes internal variables
Symbolic execution

Ex: a-b-(c-b)^2-d-f with X,Y

\[ \langle a, \langle z,1.>, \langle w,\text{abs}(Y)\rangle, \text{true} \rangle \]

\[ \langle a-b, \langle z,1.>, \langle w,\text{abs}(Y)\rangle, \text{abs}(Y) \neq 0 \rangle \]

\[ \langle a-b-c, \langle z,X>, \langle w,\text{abs}(Y)-1\rangle, \text{abs}(Y) \neq 0 \rangle \]

\[ \langle a-b-c-b, \langle z,X>, \langle w,\text{abs}(Y)-1\rangle, \text{abs}(Y) \neq 0, \text{abs}(Y)-1 \neq 0 \rangle \]

\[ \langle a-b-c-b-c, \langle z,X^2>, \langle w,\text{abs}(Y)-2\rangle, \text{abs}(Y) \neq 0, \text{abs}(Y)-1 \neq 0 \rangle \]

\[ \langle a-b-(c-b)^2, \langle z,X^2>, \langle w,\text{abs}(Y)-2\rangle, \text{abs}(Y) \neq 0, \text{abs}(Y) \neq 1, \text{abs}(Y)-2 = 0 \rangle \]

\[ \langle a-b-(c-b)^2-d, \langle z,X^2>, \langle w,\text{abs}(Y)-2\rangle, \text{abs}(Y) \neq 0, \text{abs}(Y) \neq 1, \text{abs}(Y) = 2, Y \geq 0 \rangle \]

\[ \langle a-b-(c-b)^2-d-f, \langle z,X^2>, \langle w,0\rangle, Y=2 \rangle \]
Computing symbolic states

- \(<\text{Path}, \text{State}, \text{PC}>\) is computed by induction over each statement of \text{Path}.

- When the \text{Path} conditions are unsatisfiable then \text{Path} is non-feasible and reciprocally (i.e., symbolic execution captures the concrete semantics).

  \(\text{ex: } <a-b-d-e-f, \{\ldots\}, \text{abs}(Y)=0 \land Y<0>\)

- Forward vs backward analysis:
  
  Forward → interesting when states are needed  
  Backward → saves memory space (states are not memoized)
Ex: \( a-b-(c-b)^2-d-f \) with \( X,Y \)

\( f,d: \ Y \geq 0 \)

\( b: \ Y \geq 0, \ w = 0 \)

\( c: \ Y \geq 0, \ w-1 = 0 \)

\( b: \ Y \geq 0, \ w-1 = 0, \ w \neq 0 \)

\( c: \ Y \geq 0, \ w-2 = 0, \ w-1 \neq 0 \)

\( b: \ Y \geq 0, \ w-2 = 0, \ w-1 \neq 0, \ w \neq 0 \)

\( a: \ Y \geq 0, \ \text{abs}(Y)-2 = 0, \ \text{abs}(Y)-1 \neq 0, \ \text{abs}(Y) \neq 0 \)

\( Y = 2 \)
Problems for symbolic evaluation techniques

→ Combinatorial explosion of paths

→ Symbolic execution constrains the shape of dynamically allocated objects

```c
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:

- $x < 10000$
- $x + 10^{12} > 10^{12}$

On the reals: $x \in ]0, 10000[$

On the floats: no solution!
Conversely,

```c
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:

- $X > 0$
- $X + 10^{12} = 10^{12}$

On the reals: no solution
On the floats: $X \in ]0, 32767.99\ldots[$

Symbolic execution of floating-point computations
(Botella, Gotlieb, Michel STVR 06)

Symbolic test data generation for floating-point programs
(Bagnara, Carlier, Gori, Gotlieb ICST’13, JoC 15, TOSEM 21)
Dynamic Symbolic Evaluation

- Symbolic execution of a concrete execution (a.k.a. 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, Dunbar et al. 2009)

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

3. Execute with x

4. Repeat 2

Up to a given bound
2nd ingredient: use concrete values

- Use actual values to simplify the constraint set

Flip

If( \( x_3 == x_1 \times x_2 \) ) ...

(\( x_1 = 6 \), \( x_2=7 \), \( x_3=42 \))

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

(2) Approximate solving -- add \( x_3 != 6 \times x_2 \) && \( x_1=6 \) (linear expr.)

or -- add \( x_3 != x_1 \times 7 \) && \( x_2=7 \) (linear expr.)

or -- add \( 42 != x_1 \times x_2 \) && \( x_3=42 \) (nonlinear expr.)

(3) Approximate solving -- add \( x_3 != 6 \times 7 \) && \( x_1=6 \) && \( x_2=7 \)

(4) Useless solving -- add \( 42 != 6 \times 7 \) && \( x_1=6 \) && \( x_2=7 \) && \( x_3=42 \)
Constraint solving in symbolic evaluation

Mixed Integer Linear Programming approaches
(i.e., simplex + Fourier’s elimination + branch-and-bound)

- CLP(R,Q) in ATGen
- lpsolve in DART/CUTE

SMT-solving (= SAT + Theories)

- STP in EXE and KLEE
- Z3 in PEX and SAGE

Constraint Programming techniques (constraint propagation and labelling)

- Colibri in PathCrawler
- Disolver in SAGE
- ECLAIR
Outline

Software Testing and Code-based Testing

Path-oriented exploration

Constraint-based exploration

Summary and further work
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 reachability problem

f( int i )
{
  a.    j = 100;
      while( i > 1)
  b.        { j++ ; i-- ;}
...
  d. if( j > 500)
  e.    ...

value of i to reach ... ?
Path-oriented exploration

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

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

2. Path conditions generation (via symbolic exec.)
   $j_1=100, \ i_1>1, \ j_2=101, \ i_2=i_1-1,...j_{15}=114, \ j_{15}>500$

3. Path conditions solving
   unsatisfiable $\rightarrow$ FAIL

Backtrack!
Constraint-based exploration

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

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 \( \Rightarrow \) unroll the loop 400 times \( \Rightarrow i_1 \in 401 .. 2^{31}-1 \)

No backtrack!
Assignement as Constraint

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

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

\[ \begin{array}{llll}
  i_1 = 3 \; ? & i_1 \text{ in -4..2} & \text{no} & i_1 \text{ in -5..3} \\
  i_2 = 16 & i_2 = 9 \; ? & i_2 = 7 \; ? & i_2 \text{ in 5..16} \; ?
\end{array} \]
Statements as (global) constraints

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

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

✓ Control structures: dedicated global constraints

Conditionnels (SSA)  
if D then C_1, else C_2; v_3=\phi(v_1,v_2)  \rightarrow  \text{ite/6}

Loops (SSA)  
v_3=\phi(v_1,v_2) \text{ while } D \text{ do } C  \rightarrow  \text{w/5}
Conditional as global constraint: ite/6

\[
\begin{align*}
  \text{ite}( x > 0, j_1, j_2, j_3, & \quad j_1 = 5, \quad j_2 = 18 ) \quad \text{iff} \\
  & \quad x > 0 \quad \rightarrow \quad j_1 = 5 \land j_3 = j_1 \\
  & \quad \neg(x > 0) \quad \rightarrow \quad j_2 = 18 \land j_3 = j_2 \\
  & \quad \neg( x > 0 \land j_1 = 5 \land j_3 = j_1 ) \rightarrow \neg(x > 0) \land j_2 = 18 \land j_3 = j_2 \\
  & \quad \neg( \neg(x > 0) \land j_3 = j_2 ) \rightarrow \quad x > 0 \land j_1 = 5 \land j_3 = j_1 \\
  & \quad \text{Join}( x > 0 \land j_1 = 5 \land j_3 = j_1 , \quad \neg(x > 0) \land j_1 = 18 \land j_3 = j_2 )
\end{align*}
\]
Loop as global constraint: $w/5$

$$v_3 = \phi(v_1, v_2)$$

while($Dec$)

1. **body**
   - $Dec_{V3 \leftarrow V1} \rightarrow body_{V3 \leftarrow V1} \wedge w(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow V_{new}})$
   - $\neg Dec_{V3 \leftarrow V1} \rightarrow v_3 = v_1$
   - $\neg (Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1}) \rightarrow \neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1$
   - $\neg (\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1) \rightarrow Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge w(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow V_{new}})$
   - $join(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge w(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow V_{new}}), \neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1)$
```java
f( int i ) {
    j = 100;
    while( i > 1)
    {
        j++ ; i-- ;
    }
    ...
    if( j > 500)
    {
    ...
    }
    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 the w relation

✓ It can be nested into other relations ite/6 or w/5 (e.g., nested loops \( w(\ \text{cond}_1, v_1, v_2, v_3, w(\text{cond}_2, \ldots)) \))

✓ 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) (Denmat Gotlieb Ducassé CP’2007)
Outline

Software Testing and Code-based Testing

Path-oriented exploration

Constraint-based exploration

Summary and further work
CBT (summary)

Proved concept in code-based automatic test data generation

Two main approaches:
- Path-oriented exploration (using symbolic evaluation techniques)
- Constraint-based exploration (using global constraints)

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

Mature tools (academic and industrial) exist, but problems remain for handling efficiently complex code (pointer arithmetic, transtyping, etc.), non-feasible code leading to unsatisfiable constraint systems, large data structures…
Further work

- Constraint acquisition for learning preconditions and generating satisfying test inputs (PhD G. Menguy, joint work with CEA, France)

- Initial states generation for testing optimal AI planners

- Test case execution scheduling with constraint acquisition

We are a team of researchers interested by real-world applications that lead to applied research problems. Long-term experience in technology transfer and technology adoption.