Constraint-based reachability: test input generation for C code

Arnaud Gotlieb
Certus Software V&V Centre
SIMULA RESEARCH LABORATORY
HID, Bergen, Norway
11 Feb. 2014
The Certus SFI Centre

Software Validation & Verification

Hosted by SIMULA Research Lab.

Established in Oct. 2011
8 years

www.certus-sfi.no
Outline

Introduction

Constraint-based program exploration

Euclide: An implementation for C code

Conclusions & Perspectives
Constraint-based reachability (CBR)

For a given program $P$ and location $loc$ in $P$, constraint-based reachability (CBR) is the process of determining: 1) if $loc$ is reachable and 2) inputs values for $P$, in order to reach $loc$ by using constraint solving techniques (CSP, LP, SAT, SMT, ...)

Reachability problems in infinite-state systems are undecidable in general!

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

Developed in the context of software testing (e.g., symbolic evaluation, mutation testing)

Lots of Research works and tools!
Solving CBR problems involves constraint solving

Even when adding bounds, hard combinatorial problem

Using Random Testing,

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

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
Our contribution:
Constraint-based program exploration
A CBR problem

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

    ...

d. if(j > 500)

e. ...
```

value of i to reach e?
Path-oriented exploration

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

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 \( \rightarrow \) FAIL

Even without loops, \#paths is exponential with \#decisions
Constraint-based program exploration

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

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 (Constraint Programming)

- Constraint reasoning over control structures $\rightarrow$ meta-constraints

- Requires to build **dedicated constraint solvers**:
  
  * filtering techniques, propagation queue management with priorities

  * specific meta-constraints for handling pointers and memory updates, floating-point computations, function calls, ...

  * structure-aware labelling heuristics
Viewing an assignment as a relation requires to normalize expressions and rename variables (through single assignment languages, e.g. SSA)

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

Using **classical filtering techniques over finite domains**: 

1. \[ i_1 = 3 \ ? \]
2. \[ i_1 \text{ in } -4..2 \]
3. no
4. \[ i_1 \text{ in } -5..3 \]

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

1. \[ i_2 = 16 \]
2. \[ i_2 = 9 \ ? \]
3. \[ i_2 = 7 \ ? \]
4. \[ i_2 \text{ in } 5..16 \ ? \]
Statements as 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
  ```

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

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

  **Conditionnals (SSA)**
  
  ```
  \text{if } D \text{ then } C_1; \text{ else } C_2  \rightarrow  \text{ite/6}
  ```

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

\[
\text{ite}( x > 0, j_1, j_2, j_3, \ j_1 = 5, \ j_2 = 18 ) \ \text{iff} \\
\textbullet \ x > 0 \quad \rightarrow \quad j_1 = 5 \ \land \ j_3 = j_1 \\
\textbullet \ \neg( x > 0 ) \quad \rightarrow \quad j_2 = 18 \ \land \ j_3 = j_2 \\
\textbullet \ \neg( x > 0 \land j_1 = 5 \land j_3 = j_1 ) \quad \rightarrow \quad \neg(x > 0) \land j_2 = 18 \land j_3 = j_2 \\
\textbullet \ \neg( \neg(x > 0) \land j_3 = j_2 ) \quad \rightarrow \quad x > 0 \land j_1 = 5 \land j_3 = j_1 \\
\textbullet \ \text{Join}( x > 0 \land j_1 = 5 \land j_3 = j_1 , \ \neg(x > 0) \land j_1 = 18 \land j_3 = j_2 )
\]

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

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

while (Dec)

\[ w(Dec, V_1, V_2, V_3, \text{body}) \text{ iff } \]

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

- \( \neg(\text{Dec}_{V_3 \leftarrow V_1} \wedge \text{body}_{V_3 \leftarrow V_1}) \rightarrow \neg\text{Dec}_{V_3 \leftarrow V_1} \wedge v_3 = v_1 \)
- \( \neg(\neg\text{Dec}_{V_3 \leftarrow V_1} \wedge v_3 = v_1) \rightarrow \text{Dec}_{V_3 \leftarrow V_1} \wedge \text{body}_{V_3 \leftarrow V_1} \wedge w(Dec, v_2, v_{\text{new}}, v_3, body_{V_2 \leftarrow V_{\text{new}}}) \)
- \( \text{join}(\text{Dec}_{V_3 \leftarrow V_1} \wedge \text{body}_{V_3 \leftarrow V_1} \wedge w(Dec, v_2, v_{\text{new}}, v_3, body_{V_2 \leftarrow V_{\text{new}}}), \neg\text{Dec}_{V_3 \leftarrow V_1} \wedge v_3 = v_1) \)
\textbf{f( int i )} { 
\> j = 100;
\> while( i > 1) 
\> \> { j++ ; i-- ;} 
\> ... 
\> if( j > 500) 
\> ... 
}

\textbf{w(Dec, V_1, V_2, V_3, body)} :-
\> \textbf{Dec}_{V_3 \leftarrow V_1} \rightarrow \textbf{body}_{V_3 \leftarrow V_1} \land \textbf{w(Dec, V_2, V_3, body)} 
\> \neg \textbf{Dec}_{V_3 \leftarrow V_1} \rightarrow v_3 = v_1
\> \neg (\neg \textbf{Dec}_{V_3 \leftarrow V_1} \land \textbf{body}_{V_3 \leftarrow V_1}) \rightarrow \neg \textbf{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1
\> \neg (\neg \textbf{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1) \rightarrow \textbf{Dec}_{V_3 \leftarrow V_1} \land \textbf{body}_{V_3 \leftarrow V_1} \land \textbf{w(Dec, V_2, V_3, body)} 
\> \textbf{join}(\textbf{Dec}_{V_3 \leftarrow V_1} \land \textbf{body}_{V_3 \leftarrow V_1} \land \textbf{w(Dec, V_2, V_3, body)} , 
\> \neg \textbf{Dec}_{V_3 \leftarrow V_1} \land v_3 = v_1)

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

\> \textbf{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!
→ only a semi-correct test input generation procedure

✓ Join is implemented using Abstract Interpretation operators (e.g., Interval and Polyhedral union, widening in Euclide, Difference constraints in Gatel, Congruences in JSolver)

✓ Special propagators based on linear-based relaxations
  Using Linear Programming over rationals (i.e., Q_polyhedra)
EUCLIDE: An implementation for C code
```c
void P_rad_eta()
{
    MEM_PBMORDR = PBMORDR;
    PBMORDR = 0x0;
    FM_PBMORDR = 0x0;
    if (!MSTDRD)
    {
        TPMSTRDR = 94;
        TPCODRDR = 194;
        TPIERMORDR = 1875;
        MERDR = 0;
    }
    else
    {
        if (TPCODRDR != 194)
        {
            if (TPCODRDR <= 0)
            {
                trait2_eta();
            }
            else
            {
                if (TPCODRDR <= (194 - 13))
                {
                    if (DIALRDR)
                    {
                        trait3_eta();
                    }
                    else
                    {
                        local merdr3g = TP_RDR_7R.merdr3g
                        if (((local_merdr3g & 0x0001) == 0x0001)
                        {
                            trait1_eta();
                        }
                    }
                }
            }
        }
    }
}
```
Conclusions & Perspectives
Conclusions

- Constraint Programming is a convenient and efficient tool for reasoning over imperative programs, as it enables:
  - constraint design and constraint-based program exploration;
  - **relational modelling** for reachability problems;
  - implementations are available! (e.g., EUCLIDE, PathCrawler)

- But **unsatisfiability ( UNSAT) detection** has to be improved (e.g., by combining techniques from SMT-solving)

- But **constraint solvers** are so tuned and optimized, that they cannot be easily showed bug-free, and blindly trusted!
Perspectives

- Constraint solving over floating-point computations
  (Bagnara Carlier Gori Gotlieb, ICST’2013)
  
  Collaboration with U.of Parma, Italy – PhD Thesis

- Formal certification of a consistency filtering constraint solver
  (Carlier Dubois Gotlieb, FM’12)
  
  Collaboration with INRIA, France – AURORA CertiSkatt Project
Thank you!