Constraint-based reachability: test input generation for C code

Arnaud Gotlieb

Certus Software V&V Centre
SIMULA RESEARCH LABORATORY

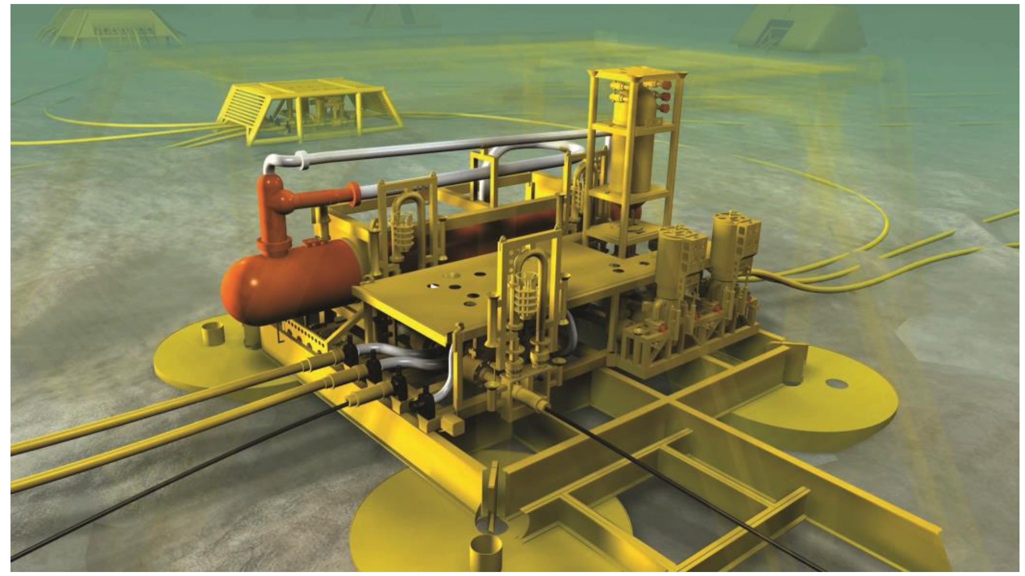
HID, Bergen, Norway
11 Feb. 2014

Certus

sfi Centre for Research-based Innovation

The Certus SFI Centre

Software
Validation&Verification

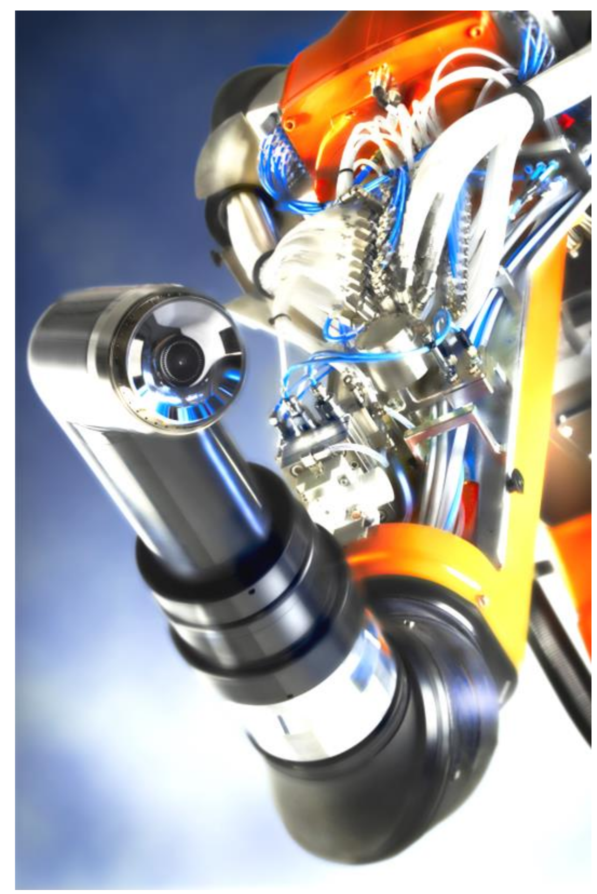


FMC Technologies

Cisco Systems Norway

Hosted by SIMULA Research Lab.

ABB Robotics
Stavanger



Norwegian Custom and excise

Established in Oct. 2011
8 years

www.certus-sfi.no

esito

Kongsberg Maritime

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

f (int x_1, int x_2, int x_3) {

if($x_1 == x_2$ && $x_2 == x_3$)

if($x_3 == x_1 * x_2$) ... }

Using Random Testing,

Prob{ reack k} = 2 over $2^{32} \times 2^{32} \times 2^{32}$ = $\mathbf{2^{-95}}$ **= 0.00000…1.**

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

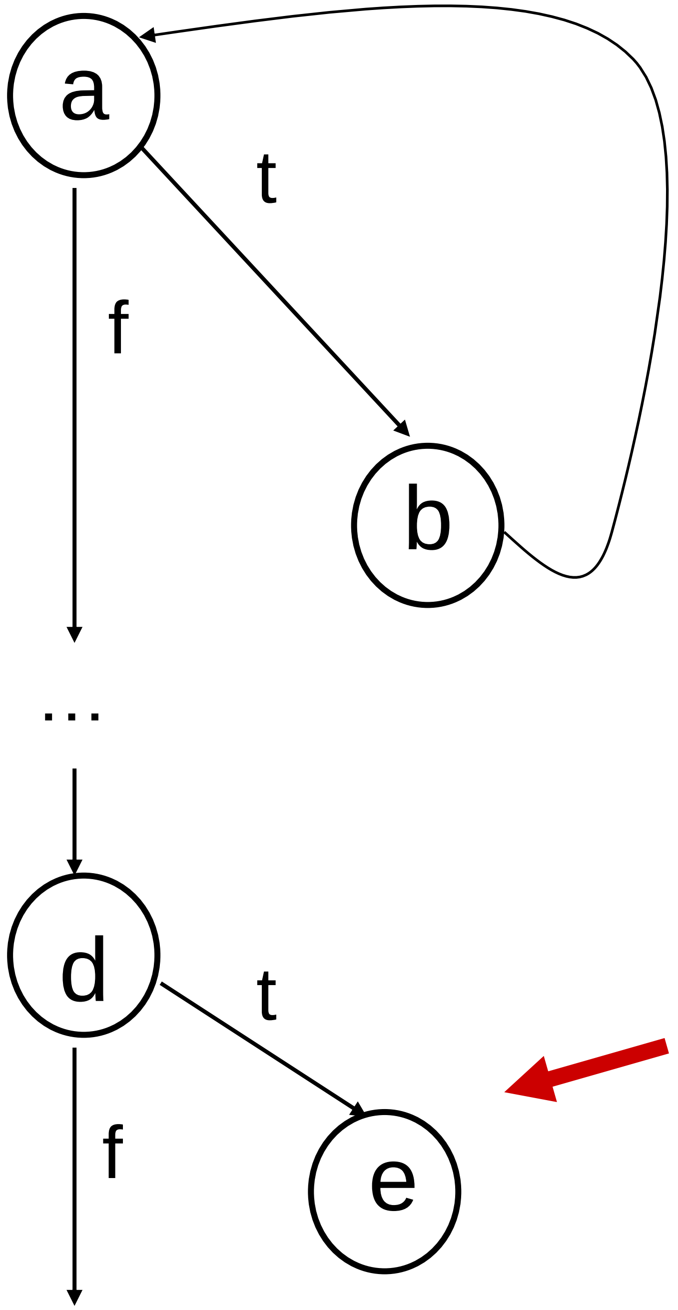
Our contribution:
Constraint-based program exploration

A CBR problem

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

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

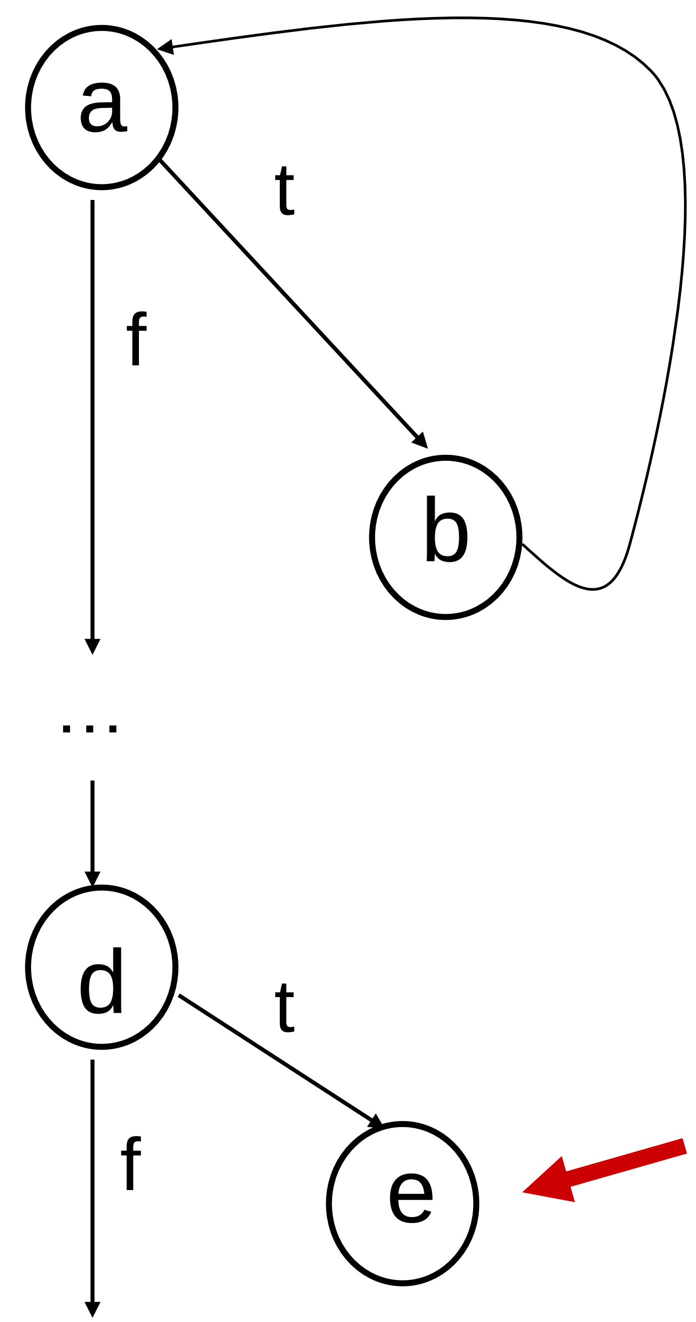
value of i to reach e ?



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., $(a\text{-}b)^{14}\text{-}\ldots\text{-}d\text{-}e$

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

3. Path condition solving
   unsatisfiable → FAIL

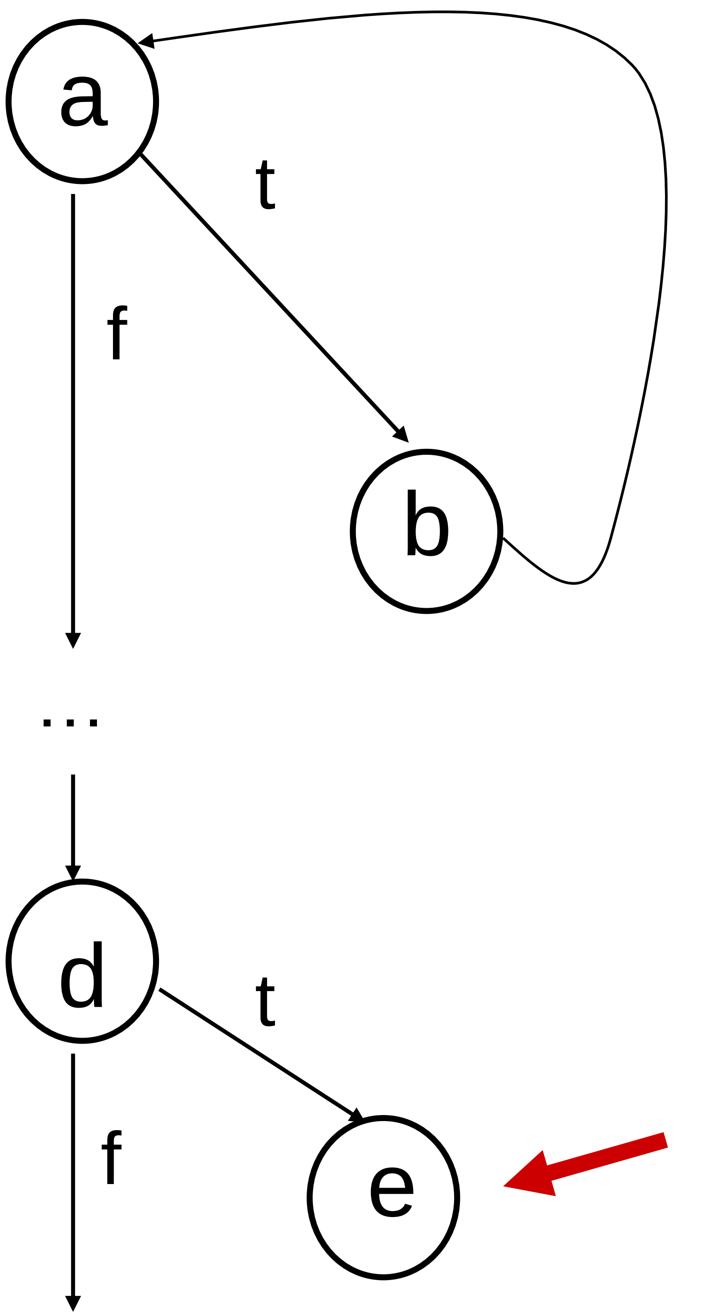
Backtrack !

Even without loops, #paths is exponential with #decisions

Constraint-based program exploration

```
  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;

   $j_1=100$, $i_3 \le 1$, $j_3 > 500$

3. Constraint model solving

   $j_1 \neq j_3$ entailed ➔ unroll the loop 400 times ➔ i_1 in 401 .. 2^{31}-1

No backtrack !

Constraint-based program exploration

- Based on a constraint model of the whole program (Constraint Programming)

- Constraint reasoning over control structures → 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, …

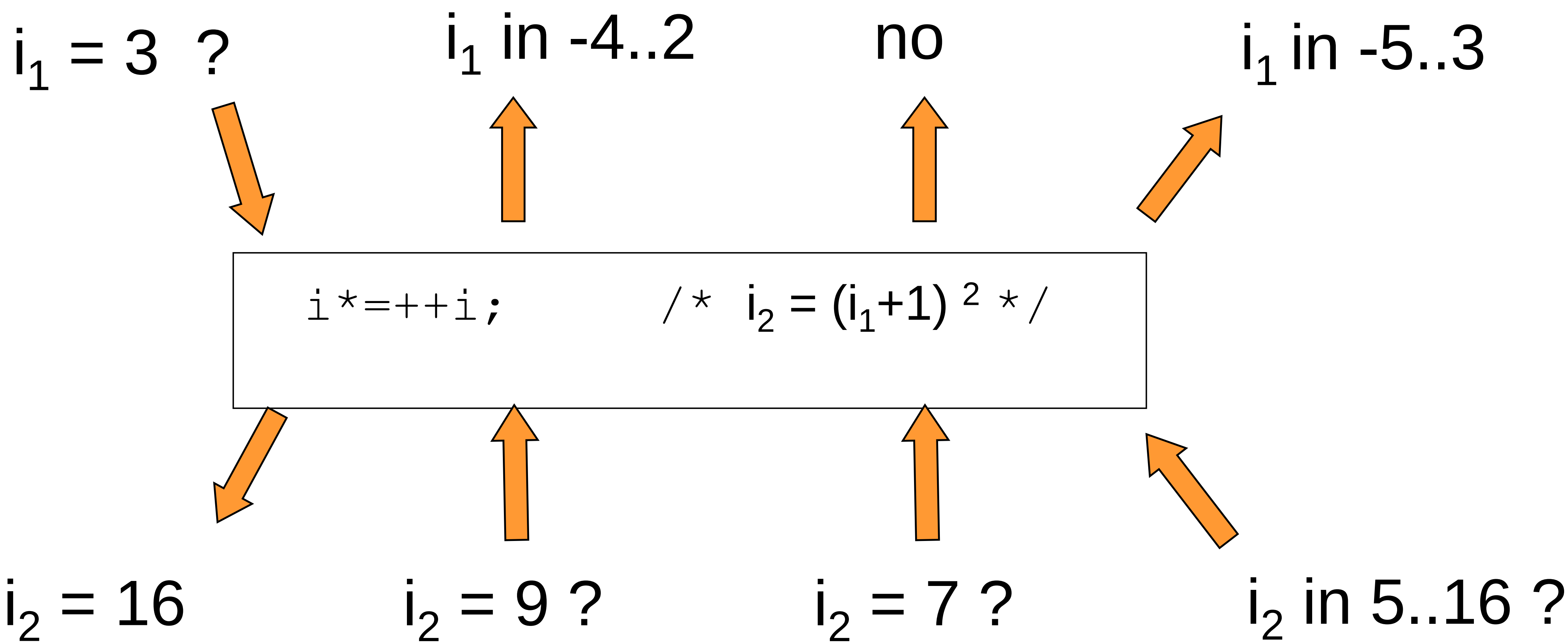
  * structure-aware labelling heuristics

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 ;` $\longrightarrow$ $i_2 = (i_1+1)^2$

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



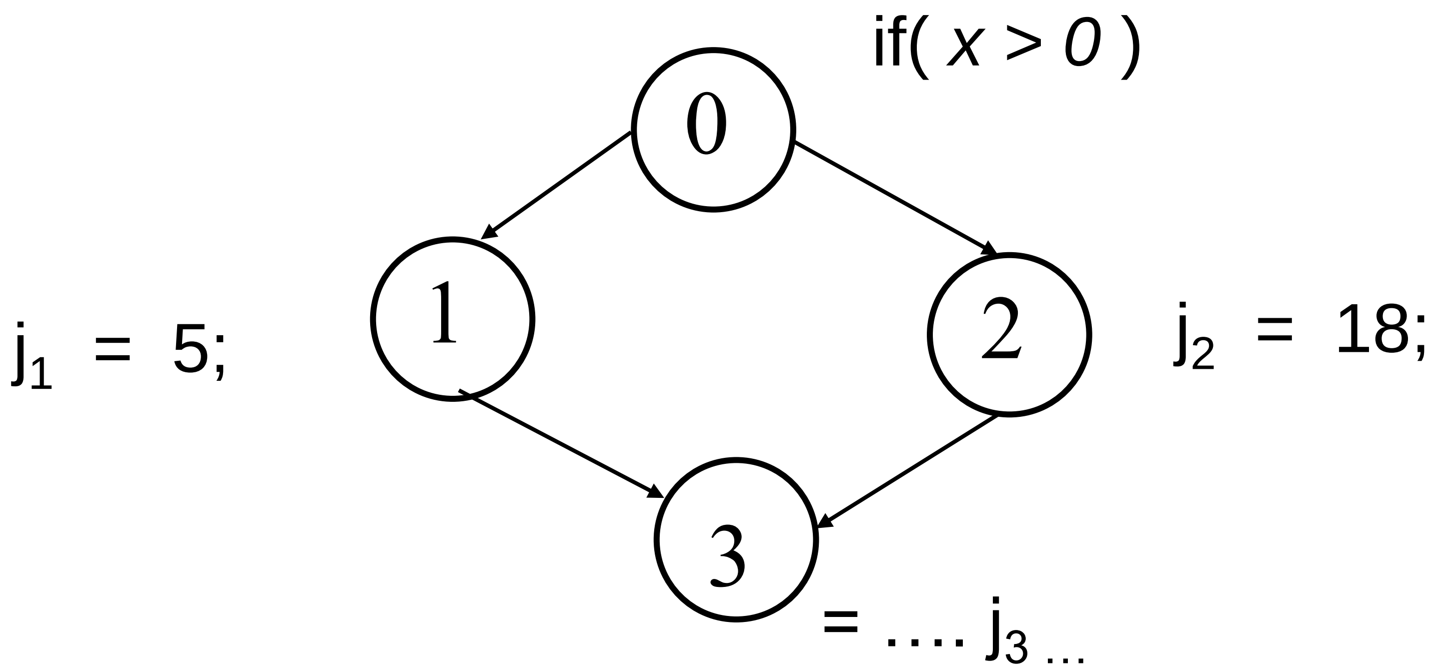
Statements as constraints

- ✓ Type declaration: `signed long x;` → x in $-2^{31}..2^{31}-1$
- ✓ Assignments: `i*=++i ;` → $i_2 = (i_1+1)^2$
- ✓ Memory and array accesses and updates:
  `v=A[i]` (or `p=Mem[&p]`) → variations of element/3
- ✓ Control structures: dedicated meta-constraints
  (interface, awakening conditions and filtering algorithms)

  Conditionnals (SSA) if D then C_1; else C_2 → ite/6

  Loops (SSA) `while D do C` → w/5

Conditional as meta-constraint: ite/6

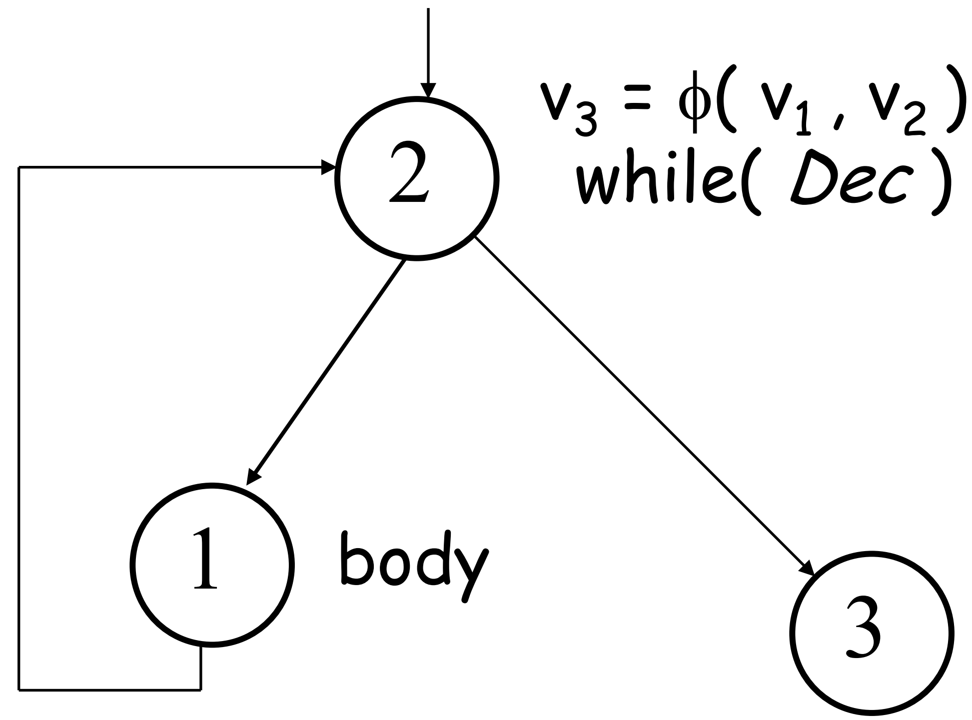


$ite(\, x > 0,\ j_1, j_2, j_3,\quad j_1 = 5,\quad j_2 = 18\,)$ iff

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

Implemented as a regular constraint
(interface, awakening conditions, filtering algo.)

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

$i = 23,\ j_1 = 100$?

no

i in $401..2^{31}-1$

$$\boxed{\mathbf{w}(i_3 > 1, (i, j_1), (i_2, j_2), (i_3, j_3),\ j_2 = j_3 + 1 \wedge i_2 = i_3 - 1)}$$

$i_3 = 1,\ j_3 = 122$

$i_3 = 10$?

$j_1 = 100$,

$j_3 > 500$?

Features of constraint-based exploration

✓ Special meta-constraints implementation for ite and w

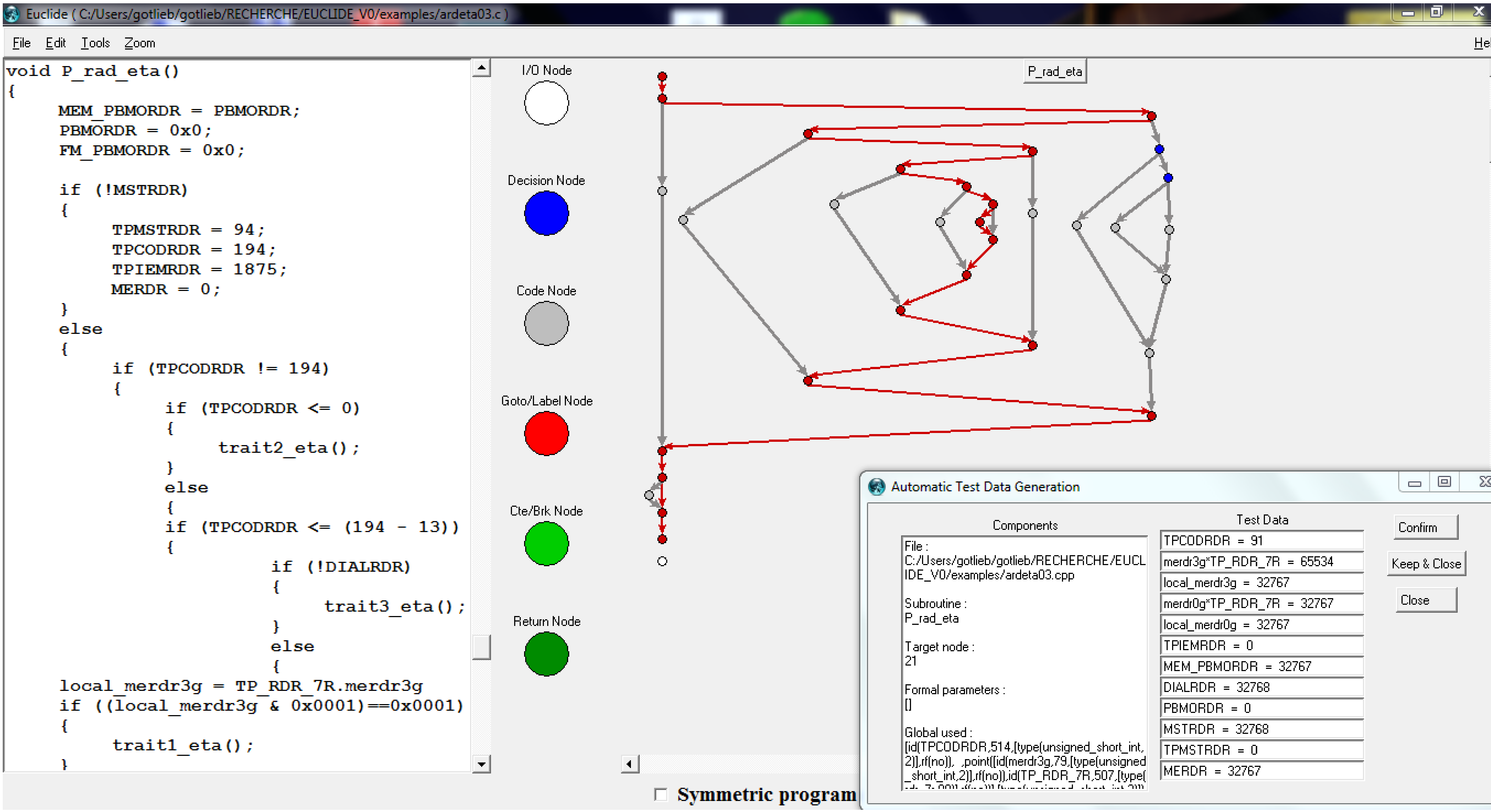
By construction, w is unfolded only when necessary
but w may NOT terminate !
→ only a **semi-correct** test input generation procedure

✓ 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

EUCLIDE



Euclide (C:/Users/gotlieb/gotlieb/RECHERCHE/EUCLIDE_V0/examples/ardeta03.c)
File Edit Tools Zoom
Help
void P_rad_eta()
{
MEM_PBMORDR = PBMORDR;
PBMORDR = 0x0;
FM_PBMORDR = 0x0;
if (!MSTRDR)
{
TPMSTRDR = 94;
TPCODRDR = 194;
TPIEMRDR = 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();
}
I/O Node
Decision Node
Code Node
Goto/Label Node
Cte/Brk Node
Return Node
P_rad_eta
Symmetric program
Automatic Test Data Generation
Components
File :
C:/Users/gotlieb/gotlieb/RECHERCHE/EUCLIDE_V0/examples/ardeta03.cpp
Subroutine :
P_rad_eta
Target node :
21
Formal parameters :
[]
Global used :
[id(TPCODRDR,514,[type(unsigned_short_int,2)],rf(no)), _point([id(merdr3g,79,[type(unsigned_short_int,2)],rf(no)),id(TP_RDR_7R,507,[type(
Test Data
TPCODRDR = 91
merdr3g*TP_RDR_7R = 65534
local_merdr3g = 32767
merdr0g*TP_RDR_7R = 32767
local_merdr0g = 32767
TPIEMRDR = 0
MEM_PBMORDR = 32767
DIALRDR = 32768
PBMORDR = 0
MSTRDR = 32768
TPMSTRDR = 0
MERDR = 32767
Confirm
Keep & Close
Close

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 reacheability 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!