

Using SICStus Prolog to build automatic test data generation tools

Arnaud Gotlieb
INRIA Rennes, France

SweConsNet'10, 05/21/10



Outline

- Automatic test data generation (ATDG)
- Using clpfd in ATDG
- Building an hybrid constraint solver (clpfd, clpq)
- Further work

Automatic test data generation (ATDG)

- Unit testing of sequential programs (e.g., a C function or a Java method) requires the generation of test input that reach given location

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

     ...
d.   if( j > 500) ← Value of i to reach e ?
e.     ...
```

- Constraint-Based Testing are techniques that address this problem with constraint solving approaches, including CP techniques

Constraint-Based Testing tools for ATDG

InKa (Gotlieb, Botella, Rueher CL'2000)	}	Sicstus 3
FPSE (Botella, Gotlieb, Michel STVR'2006)		
Euclide (Gotlieb ICST'2009)		
FocalTest (Carlier, Dubois, Gotlieb ICSoft'2010)		Sicstus 4

Built in SICStus Prolog, using **clpfd** at different levels

(Carlsson M., Ottosson G., Carlson B. *An Open-Ended Finite Domain Constraint Solver*, PLILP'1997).

Bound- and domain- consistencies, entailment checking, global constraint interface, reification and new labelling heuristics

Euclide: an hybrid constraint solver

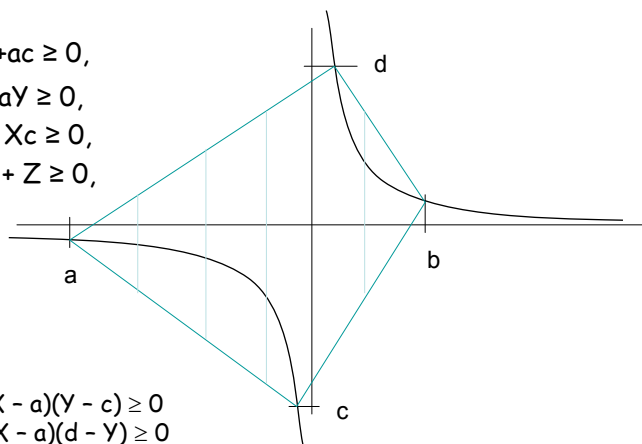
- Constraint propagation queue management
 - 2 priority levels: arithmetical constraints \rightarrow high
 - global constraints (reified, ite, w, ...) \rightarrow low
- Hybridation in Euclide comes in two distinct flavors
 - \rightarrow Integer (FD var) and rational variables (p/q)
 - \rightarrow Constraint Propagation (clpfd) and Linear Programming (Holzbaur's clpq)
- Maintains coherence through the notion of
 - Dynamic Linear Relaxation (DLR),
 - i.e., to build an over-approximation (Q_Polyhedron) for each constraint

DLR of multiplication

McCormick 76

$$Z = X * Y, \quad X \text{ in } a..b, Y \text{ in } c..d$$

$$\Leftrightarrow \{ \begin{array}{l} Z - Ya - Xc + ac \geq 0, \\ Xd - Z - ad + aY \geq 0, \\ bY - bc - Z + Xc \geq 0, \\ bd - bY - Xd + Z \geq 0, \\ a \leq X \leq b, \\ c \leq Y \leq d \end{array} \}$$



A consequence of $(X - a)(Y - c) \geq 0$
 $(X - a)(d - Y) \geq 0$
 ...

DLR of reification

- Reification associates a boolean var. to an expression

$$R = (X \leq Y) \quad \text{where } X \text{ in } a..b, Y \text{ in } c..d \text{ and } R \text{ in } 0..1$$

$$\Leftrightarrow \{ Y - X + 1 - (1 - a + d) * R \leq 0, \quad (X - Y) - (b - c) * (1 - R) \leq 0 \}$$

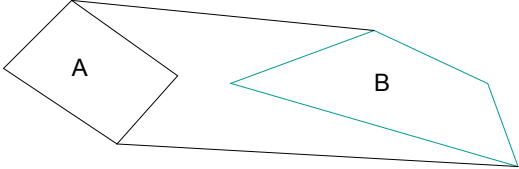
$$\begin{array}{ccccccc}
 & \nearrow & & \nwarrow & \nearrow & & \nwarrow \\
 1 - F(X,Y) & \leq & 1 - \text{Min}(F(X,Y)) * R & \leq & \text{Max}(F(X,Y)) * (1-R) \\
 R = (F(X,Y) \leq 0) & & R = (F(X,Y) \leq 0) & & & &
 \end{array}$$

DLR of *control structures ite/6, w/5, ...*

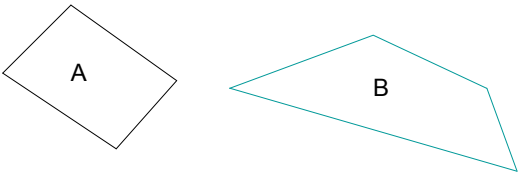
- Based on abstract domains operators
- For join in conditionals and loops
 - weak-join operator [Sankaranarayanan VMCAI'06]
 - based on calls to the simplex and branch-and-bound (clpq library)
 - no Fourier's elimination step required
- For loops → widening operators from both the Interval and Polyhedral abstract domains [Denmat Gotlieb Ducassé CP07]

Weak_join operator

convex hull computation

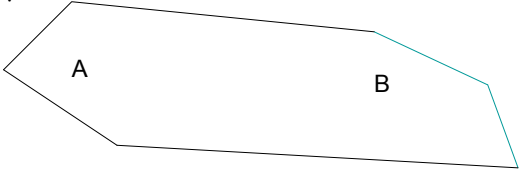


Weak join

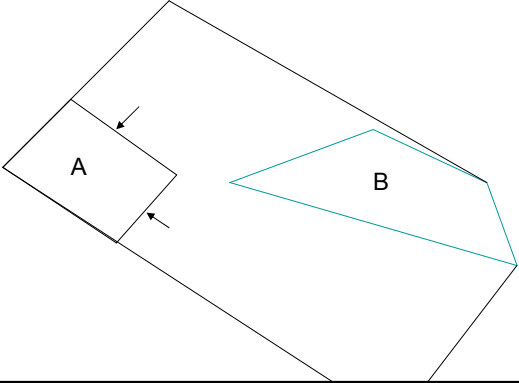


Weak_join operator

convex hull computation

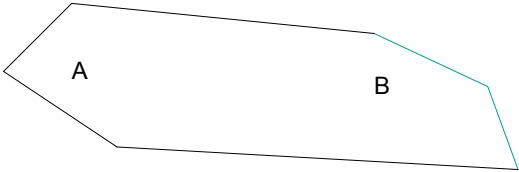


Weak join

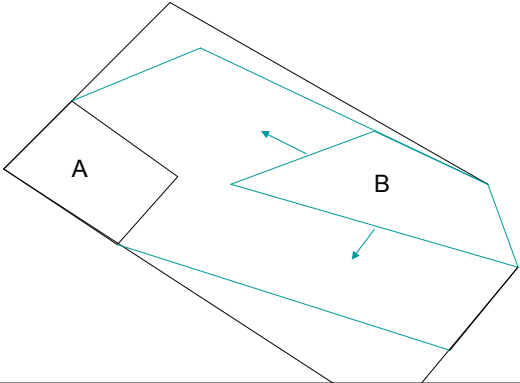


Weak_join operator

convex hull computation

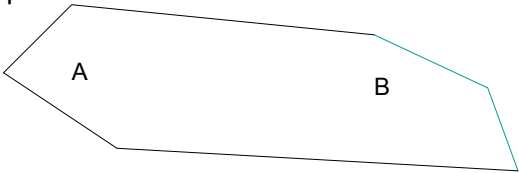


Weak join

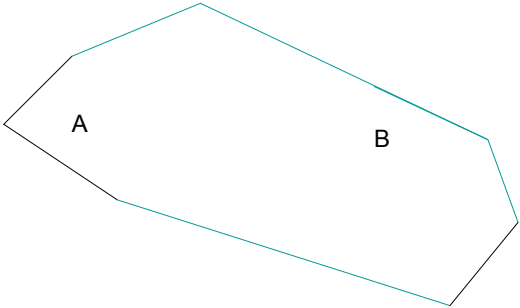


Weak_join operator

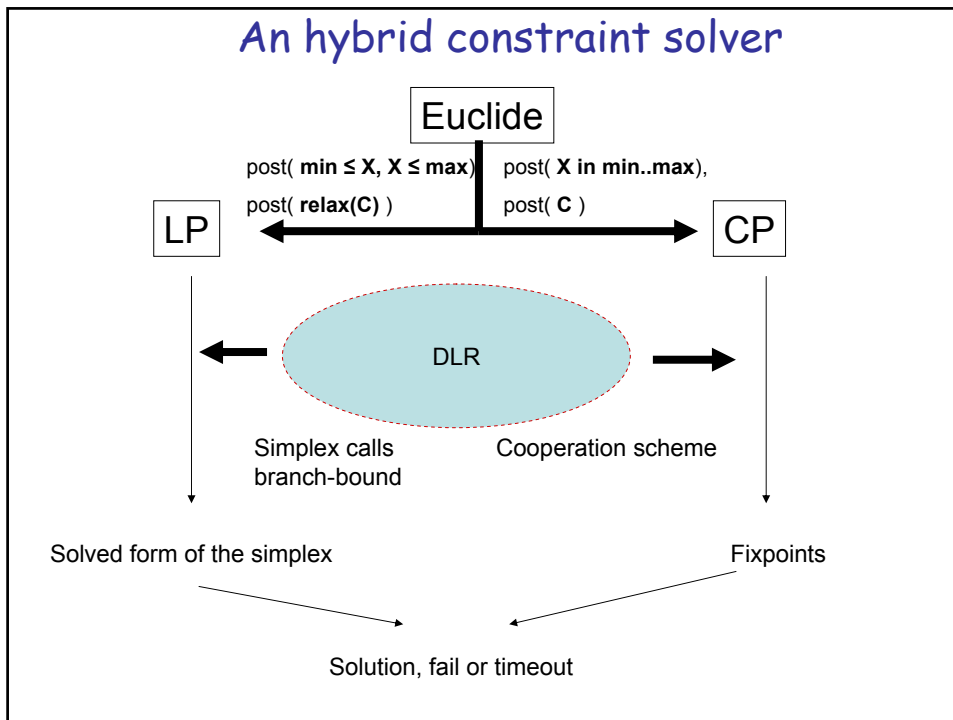
convex hull computation



Weak join



An hybrid constraint solver



A simple cooperating scheme

```

solve_clpfd(ENV) :-
    is_void(ENV),
    !,
    call_q_solver(ENV).
solve_clpfd(ENV) :-
    pop_high_priority(ENV, CONT),
    solve_cont(CONT, ENV),          /* call clpfd, compute DLR */
    solve_clpfd(ENV).

call_q_solver(ENV) :-
    get_atts(ENV, [+clpq(Q)]),
    call_clpq(ENV, Q),
    get_atts(Q, +movement(Move)),
    (Move == yes -> solve_clpfd(ENV) ; true ).
    
```

Pros/Cons

- Control over the propagation queue (priorities, awakening conditions, structure-aware heuristics)
- Hybridation (i.e., Linear Programming) permits to solve linear over-approximation of the store during initial propagation

e.g., ?- domain([X, Y], -32768, 32767), Z #= X*Y, Z #= X + Y, Z #> 4.

Clpfd: X,Y in -32762..32767, Z in 5..32767

Euclide: fail

- But, does not cut propagation cycles that are responsible of slow convergence phenomena, e.g.,

?- domain([X, Y], -2²⁵, 2²⁵), X #= Y+1, X #= Y.

clpfd, Euclide: ???

Concluding remarks

- SICStus Prolog with its constraint libraries is a very good choice for implementing constraint-based automatic test data generator

if we have control over the constraint propagation queue

- clpfd / clpq are powerful tools to implement integer constraint solving

if there is no value overflow (in SICStus 3, FD values are encoded on 26 bits)

- Our experiments (not shown in this presentation) show that Euclide is competitive with dedicated automatic test data generators (e.g., PEX that uses the SMT-solver Z3)

Thanks to SICStus clpfd...

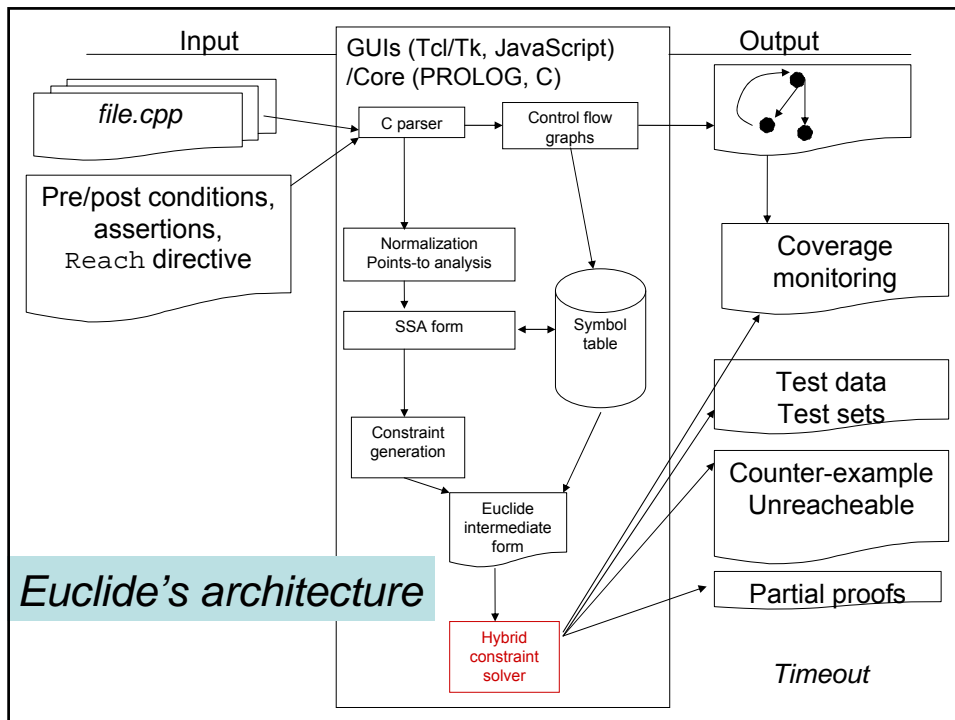
<http://euclide.gforge.inria.fr/>

Further work

- Explore the use of other abstract domains such as
 - the Octagons domain (e.g., $\pm X_i \pm X_j \leq c$) [Miné's Thesis 2001, HOSC'06]
 - the Congruence domain (e.g., $X \equiv a \pmod{b}$)
- Explore the combination of SMT-solving and clpfd

Seeing clpfd as another decision procedure, combining clpfd with a congruence closure algorithm

Thanks !



Constraint propagation

- Each constraint C implements a **filtering operator** p that maps domains to domains
- The filtering operator must be
 - correct** ($a \in D \wedge C(a) \Rightarrow a \in p(D)$) and **decreasing** ($p(D) \subseteq D$)
- Given a domain D , and a set of filtering operators $\{p_1, \dots, p_n\}$, a constraint propagation solver computes :

$$\text{solv}(\{p_1, \dots, p_n\}, D) = \text{gfp}(p_1 \circ \dots \circ p_n)(D) \quad [\text{Schulte Stuckey CP'04}]$$

- Filtering operators usually exploit several operations over the domains:
 - satisfiability testing $\text{Is } p(D) = \emptyset ?$
 - projection $p(D)|_x$
 - intersection $p(D \cap D')$
 - union $p(D \cup D')$
 - arithmetic and comparison operators

Denmat's Thesis

- Computing $\text{gfp}(p_1 \circ \dots \circ p_n)(D)$ can be understood as computing over an abstract domain [Denmat's PhD thesis 08]

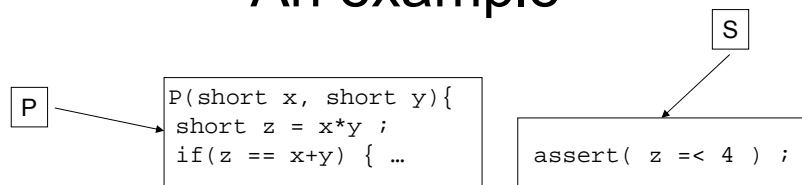
- $\alpha : D \rightarrow D^\#$
- $p_i^\# : D^\# \rightarrow D^\#$
- $\text{gfp}(p_1^\# \circ \dots \circ p_n^\#)(\alpha(D))$ is an over-approximation of the solution set

- Filtering operators computes local consistencies :

Bounds consistency corresponds to the Interval Abstract Domain

LIB-consistency [Solnon 97] corresponds to the Polyhedral Abstract Domain

An example



Constraint propagation on $\neg S \wedge P$: $Z = X * Y, Z = X + Y, Z > 4$ yields only to

$$z \in 5 \dots 32767 \quad \text{and} \quad x, y \in -32764 \dots 32767$$

and then a costly labelling process starts!

```
P(short x, short y) {
  short z = x*y ;
  if(z == x+y) {... assert( z =< 4 ) ;
```

X in -10 .. 10, Y in -10 .. 10, Z in -10 .. 10, Z = X*Y, Z = X+Y, Z > 4

↓ Constraint propagation

X in -7 .. 10, Y in -7 .. 10, Z in 5 .. 10, Z = X*Y, Z = X+Y, Z > 4

↓ $7X + 7Y + Z + 49 \geq 0$
 $10X - 7Y - Z + 70 \geq 0$ + Simplex calls on
 $-7X + 10Y - Z + 70 \geq 0$ X,Y,Z bounds
 $-10X - 10Y + Z + 100 \geq 0$

X in -2 .. 9, Y in -2 .. 9, Z in 5 .. 10, Z = X*Y, Z = X+Y, Z > 4

↓ Constraint Propagation
 $2X + 2Y + Z + 4 \geq 0$
 $9X - 2Y - Z + 18 \geq 0$ + Simplex calls on
 $-2X + 9Y - Z + 18 \geq 0$ X,Y,Z bounds
 $-9X - 9Y + Z + 81 \geq 0$

X in 0 .. 8, Y in 0 .. 8, Z in 5 .. 10, Z = X*Y, Z = X+Y, Z > 4

```
P(short x, short y){
  short z = x*y ;
  if(z == x+y) {... assert( z =< 4 ) ;
```

...
 X in 0 .. 8, Y in 0 .. 8, Z in 5 .. 10, Z = X*Y, Z = X+Y, Z > 4

↓ Constraint Propagation

X in 1 .. 8, Y in 1 .. 8, Z in 5 .. 10, Z = X*Y, Z = X+Y, Z > 4

↓ $-X - Y + Z + 1 \geq 0$
 $8X + Y - Z - 8 \geq 0$
 $X + 8Y - Z - 8 \geq 0$
 $-8X - 8Y + Z + 64 \geq 0$

X in 1 .. 8, Y in 1 .. 8, Z in 5 .. 9, Z = X*Y, Z = X+Y, Z > 4

↓ ...

Fail Z = X*Y, Z = X+Y, Z > 4

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

w(Dec, V₁, V₂, V₃, body) :-

- Dec_{V₃←V₁} → body_{V₃←V₁} ∧ w(Dec, V₂, V_{new}, V₃, body_{V₂←V_{new}})
- ¬Dec_{V₃←V₁} → V₃=V₁
- ¬(Dec_{V₃←V₁} ∧ body_{V₃←V₁}) → ¬Dec_{V₃←V₁} ∧ V₃=V₁
- ¬(¬Dec_{V₃←V₁} ∧ V₃=V₁) → Dec_{V₃←V₁} ∧ body_{V₃←V₁} ∧ w(Dec, V₂, V_{new}, V₃, body_{V₂←V_{new}})
- join(Dec_{V₃←V₁} ∧ body_{V₃←V₁} ∧ w(Dec, V₂, V_{new}, V₃, body_{V₂←V_{new}}, ¬Dec_{V₃←V₁} ∧ V₃=V₁)

i = 23, j₁=100 ? no i in 401..2³¹-1

w(i₃ > 1, (i₁,j₁), (i₂,j₂), (i₃,j₃), j₂ = j₃ + 1 ∧ i₂ = i₃ - 1)

i₃ = 1, j₃ = 122 i₃ = 10 ? j₁ = 100, j₃ > 500 ?

InKa: using the SICStus clpfd global constraint definition interface

Interface: Input-Output variables of the relation

```
clpfd:dispatch_global(+Constraint, +S0,-S1, -Actions)
where Actions can be exit, fail, X=V, X in D, call(Goal)
```

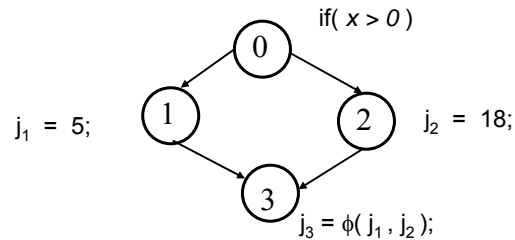
Awakening conditions

```
fd_global(:Constraint, +S, +Susp)
where Susp can be val, dom, min, max, minmax
```

Filtering algorithm

Any Prolog code,
can be defined with a set of guarded-constraints $C_1 \rightarrow C'_1, \dots, C_n \rightarrow C'_n$
(variations of blocking-ask and CHRs)

Conditional as global constraint: ite/6



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

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

A simplified implementation of ite/6

```

ite(C, V0,V1,V2, THEN, ELSE) :-
    add_dom(V0-V1-V2, DOM),
    clpfd:fd_global(ite_c(C, V0-V1-V2, THEN,ELSE),st, DOM).

clpfd:dispatch_global(ite_c(C,V0-V1-V2,THEN,ELSE),st,st, Actions):-
    ite_solve(C,THEN,ELSE, V0-V1-V2, Actions).

ite_solve(C, THEN, _, _, Actions) :-
    is_entailed(C),
    !, Actions = [exit, call(THEN)].

ite_solve(C, _, ELSE, _, Actions) :-
    is_disentailed(C),
    !, Actions = [exit, call(ELSE)].

ite_solve(C, THEN, ELSE, _-_-V2, Actions) :-
    \+(call((neg(C),ELSE,assert(V2))),!), Actions=[exit,call(C-THEN)].

ite_solve(C, THEN, ELSE, _-V1-, Actions) :-
    \+(call((C,THEN,assert(V1))),!), Actions=[exit,call(neg(C)-ELSE)].

ite_solve(C,THEN,ELSE, V0-_-_, Actions) :-
    !, Actions = [retract((V1,V2)),call(join(C,THEN,ELSE,V0-V1-V2))].
    
```