About CP and Test Generation

and also arrays and bitvectors

Sébastien Bardin
CEA LIST (Paris, France)

CP Meets CAV

A few words of context

CEA is a French large multi-disciplinary research center

nuclear power, astrophysics, quantum physics, biology / biotechnologies, neuro-sciences, secret military activities, nano-technologies, green energies, computer science, ...

Formal methods at CEA

- 3 labs, $\approx$ 60 full-time researchers
- strong focus on industrial applications
- model-based engineering (design & verif.), program verification and testing
- a few industrial success stories
  - Frama-C (static analysis of safety properties, C code)
  - Gatel (CP-based test generation, Scade models)
  - Fluctuat (static analysis for numeric precision, C code)

Constraint Programming at CEA LIST

CP used for test data generation

while our verification tools rely on AI or SMT

Three different tools at different levels

- Gatel : SCADE models
- PathCrawler : C programs
- Osmose : executable files

One common CP(FD) solver : COLIBRI [mainly B. Marre & B. Blanc]

- bounded ints, floats, bitvectors, arrays
- tailored at proving satisfiability (and produce witness)
- CP framework dedicated to software analysis
  - huge domains, many unsat or hard-to-solve formulas

Test data generation over SCADE programs

- ✓ simple high-level control structure : 1 big loop
- ✓ mostly simple data types : int, bool, read-only arrays
- ✓ no memory heap, pointers, dynamic allocation
- × reactive system : test input = (long) sequences of input values
- × floats commonly used
- × tricky synchronization mechanisms (modes, priorities, clocks)

Goal-oriented generation, global view of the program

- cf. Arnaud yesterday
- the tool can answer : yes (+ witness), no, ... or timeout

Gatel (2)

Technology

- COLIBRI
- + constraints dedicated to SCADE "control" operators
  - overapprox. of `ite` / `loop`, lazy unrolling
  - necessary to efficiently handle clocks (SCADE V6)
- + discovery of affine relationships to infer smart unrolling
  - relates part of constraints to # iter
  - allow to infer a lower bound on the necessary # iter

Applications

- used internally by the nuclear certification authority (IRSN)
- other uses in aeronautics
- feedback : clearly beats (at least one) commercial bmc tool on the customer's class of problems

PathCrawler & OSMOSE

(Unit) test data generation over embedded systems

- C for PathCrawler, binary-level for Osmose
- middle-size programs (up to 5000-10000 loc), loops, functions
- rather simple data types
- yet floats, arrays, memory heap, pointers
- OSMOSE : bitvector operations, dynamic jumps

Forward path exploration, local view of the program

- output : set of (test inputs, path) + coverage measure

PathCrawler & OSMOSE (2)

Technology

- Dynamic Symbolic Execution [Sage, Pex, Klee, Exe, Cute, etc.]
  - combine concrete execution and symbolic reasoning
  - advantages : prune infeasible paths asap, provide correct underapproximations if required
  - search and solving have mostly no interaction here
- Constraints : COLIBRI + arrays + memory model + BV
- Path search : dedicated search and pruning techniques

Some industrial applications

- unit testing of embedded systems (aeronautics, automative industry)
- wcet estimation (PathCrawler)
- completion of a customer test suite (OSMOSE)

Outline

COLIBRI : an efficient CP solver for large domains

COLIBRI [Marre-Blanc 05] is a CP(FD) solver for large domains

Abstract domains in order to tackle huge state space

- unions of intervals (Is) + congruence (C)
  $[X \in [0..5], [100..200] \wedge X \equiv 0[5]]$
- propagations on the two domains deeply interwoven (Is+C)
- more precision than just Is $\times$ C
- Is useful for case-splits [and modular arith. and arrrays]

Local propagators for all standard arithmetic operations

- $+$, $-$, $\leq$, $<$, $=$, $\neq$, $\times$, $/$, mod, ^{2}, $\sqrt{}$, $\|\cdot\|$, min, max

Deductive rules for limited symbolic deduction

Global (incomplete) reasoning through difference logic

Domains and local propagators

```
arith_plus(A,B,R) :-      // A+B=R
(
  integer(A,B)? R == eval(A+B) , success
;
  D(R) ← D(R) ∩ (D(A) + D(B)),
  D(A) ← D(A) ∩ (D(R) - D(B)),
  D(B) ← D(B) ∩ (D(R) - D(A)),
  wait(...)
)
```

Define $+, -$ over Is+C

- easy for intervals
- a bit more tricky for congruences (involves pgcd)

A few deduction rules

Backup domain reasoning with simple syntactic deductions

Use a few simple rewriting rules

- local rules [e.g. identity elt.]
- or more global (FC,C), but no complete reasoning
- the main goal is to deduce some $x = y$

Not preprocess only, included inside the propagator

```
arith_minus(A,B,R) :-      // A-B=R
(
  integer(A,B)? R == eval(A-B) , success
;
  B==0? R==A, success
;
  A==B? R==0, success
;
  ...,
```

Global constraints

Local reasoning is not enough : $x < y$ and $y < x$

- assume $D_x = D_y = [0..100]$
- propagation, step 1 : $D_x = [0..99], D_y = [1..100]$
- propagation, step 2 : $D_x = [2..99], D_y = [1..98]$
- propagation, step 3 : $D_x = [2..97], D_y = [3..98]$
- ...
- UNSAT

steps linear in the size of domains = BAD !

Global constraints (2)

Global (incomplete) reasoning through difference logic

- $\bigwedge x_i - y_i \leq k$ [reason over integers, $O(n^3)$]
- answer : unsat, or better bounds

Lazy scheme, deep integration with local propagation

- local propagators can send new formulas to Δ, and query it
  - $x - y \leq z$
  - z becomes instantiated to 100
  - send $x - y \leq 100$ to Δ
- Can be expensive
  - lots of algorithmic tuning, and probably black magic

A few examples

Assume $D_A = D_B = [0..100]$

$A + B = R$: then $R \in [0..200]$ // only I

$5 \times A = R$: then $R \in [0..500], R \equiv 0[5]$ // needs I+C

$A + 0 = R$: then $A = R$ // deduction rules

$A < B \wedge B < A$: then unsat // global constraint

What else ?

Floating-point arithmetic [a story for another day]

- rely on the integer domain, dedicated propagators
- high-level view of floats, no bitblasting

Bitvectors : see after

- rely on the integer domain, plus a new domain
- high-level view of bv, no bitblasting

Arrays : see after

- ongoing work
- high-level communication between a symbolic decision procedure and CP

What else ? (2)

Quantifiers, UF : no

- UF not so useful in test generation
- quantifers would be nice for precond/postcond, but need witness

$\vee$, $\Rightarrow$, $\Leftrightarrow$: yes, but nothing fancy there

- Gatel has its own features
- PathC & OSMOSE do not really need that

Search : not part of COLIBRI, each tool has its own

Conclusion about CP and verification

CP can be tuned for real-size verification problems

- at least for testing
- ✓ rather good at proving sat, and produce a witness !
- × but not very good for valid and $\Rightarrow$

Beware : forget about any propagation based on concrete enumeration (can be hidden in learning, global constraints, etc.)

A few interesting features of CP

- approximation of disjunctive constraints
- dynamic addition of constraints, controlled from the propagators
- easy to quickly support a new constraint : needs only a checker

Outline

COLIBRI

CP and bitvectors

CP and arrays

About CP and verification

The theory of bit-vectors

Variables range over arrays of bits

Common operations

- bitwise : $\sim, \&, |, xor$
- arithmetic : $\oplus, \ominus, \otimes, \oslash_u, \oslash_s, \%_u, \%_s$
- relations : $=, \neq, \leq_u, <_u, \leq_s, <_s$
- shifts : $\ll, \gg_u, \gg_s$
- extensions : $ext_u(A, k), ext_s(A, k)$
- concatenation : $A :: B$
- extraction : $A[i..j]$

Bit-blasting

Bit-blasting : standard way to solve problems over BV

- encode BV formula into an equisatisfiable boolean formula

Overwhelming advantage : rely on the efficiency of SAT solvers

- small effort for good performance
- easy integration into SMT solvers [Stp,Boolector,MathSat,etc.]

Shortcomings

- formula explosion : too large boolean formulas on some "arithmetic-oriented" BV-formulas
- no more information about the BV-formula structure : may miss high-level simplifications

Our approach : word-level CP-based BV solving

Goals

- develop a decent CP-based BV solver
- no bit-blasting (word-level approach)
- see how word-level approach compare wrt. low-level approach

Word-level approach

- reason on bit-vectors rather than on their separate bits
- BV variables are encoded into bounded integer variables
- BV operators are seen as integer arithmetic operators

Contribution

Difficulty

- word-level CP-based approaches already tried
  [Diaz-Codognet 01, Ferrandi-Rendine-Sciuto 02]
- performance very far from SAT-based approaches
  [Sülflow-Kühne+ 07]

Existing works rely on standard CP(FD)

- for small domains and/or linear integer arithmetic
- does not fit the needs of word-level BV solving

Our results

- a new CP(BV) framework dedicated to BV solving
- fill the gap with the best SAT approaches
- better scaling than SAT approaches w.r.t. BV sizes

Direct word-level encoding : examples (1)

Each bit-vector A is encoded by its unsigned integer value $[\![A]\!]$
Bit-vectors operators are encoded by common integer operators

- (cheap) A $\leq$ B
  - becomes $[\![A]\!] \leq [\![B]\!]$
- (cheap) A :: B = R
  - becomes $[\![A]\!] \times 2^{size(B)} + [\![B]\!] = [\![R]\!]$
- (cheap) $\sim$ A = R
  - becomes $2^N - 1 - [\![A]\!] = [\![R]\!]$

Direct word-level encoding : examples (1)

Each bit-vector A is encoded by its unsigned integer value $[\![A]\!]$
Bit-vectors operators are encoded by common integer operators

- (expensive) A $\oplus$ B = R
  - becomes $([\![A]\!] + [\![B]\!])\ mod\ 2^N = [\![R]\!]$
  - introduce modulo
- (expensive) ext_s(A,k) = R
  - become $R = ite(([\![A]\!] < 2^{N-1})?\ [\![A]\!] : [\![A]\!] + 2^k - 2^{size(A)})$
  - introduce case-split

Direct word-level encoding : examples (1)

Each bit-vector A is encoded by its unsigned integer value $[\![A]\!]$
Bit-vectors operators are encoded by common integer operators

- (very expensive) A & B = R
  - perform bit-blasting, introduce strong linear relationships
  - introduce A_is, B_is and R_is in $\{0, 1\}$
  - $R_1 = min(A_1, B_1) \wedge \ldots \wedge R_n = min(A_n, B_n)$
    $\wedge \sum A_i \cdot 2^{i-1} = [\![A]\!] \wedge \sum B_i \cdot 2^{i-1} = [\![B]\!] \wedge \sum R_i \cdot 2^{i-1} = [\![R]\!]$

Direct encoding into CP does not work

We tried direct encoding into COLIBRI : very bad !

Local propagation does lots of useless work because of nested operators

- typically : a congruence is propagated through $+$, then destroyed by *mod*

Explicit *ite* propagations are too approximated

- we are on very particular cases

Non-linear arithmetic everywhere : difference-logic is useless

Almost nothing on bitwise operations

- do not think of bitblasting for word-level approach

With this encoding, COLIBRI performs better then most propagations are turned off ! !

Dedicated features for bitvectors inside COLIBRI

Dedicated propagators for Is/C domain

- no explicit case-split or modulo
- better propagation, no waste propagation

The new domain BV and its propagators

- no bit-blasting on bitwise operators
- efficient propagation on most "linear bitwise" operations

Propagators to share information between BV and Is/C

Specific deduction rules to reduce BVA to IA

- benefits from difference-logic and other COLIBRI optims

Dedicated Is/C propagators

Dedicated propagators for Is/C domain

- no introduction of additional variables
- no introduction of "modulo" operation everywhere
- signed operations handled without explicit case-split

For A $\oplus$ B = R

Define basic propagation $\oplus_I$ from I $\times$ I $\mapsto$ Is by

$$[m_1..M_1] \oplus_I [m_2..M_2] \mapsto$$
$$\begin{array}{ll} [m_1 + m_2..M_1 + M_2] & \text{if } M_1 + M_2 < 2^N \\ [m_1 + m_2 - 2^N..M_1 + M_2 - 2^N] & \text{if } m_1 + m_2 \geq 2^N \\ [m_1 + m_2..2^N - 1] \cup [0..M_1 + M_2 - 2^N] & \text{otherwise} \end{array}$$

Unions of intervals are required here

Dedicated Is/C propagators

Dedicated propagators for Is/C domain

- no introduction of additional variables
- no introduction of "modulo" operation everywhere
- signed operations handled without explicit case-split

For R = SignExt(A,N') // initial size N, $N' \geq N$
Signed Extension : $\mathsf{I} \times \mathbb{N} \mapsto \mathsf{Is}$:

// Mask is $(2^{N'} - 1) - (2^N - 1)$

$[m_1..M_1] \mapsto$

$[m_1..M_1]$ if $m_1, M_1 < 2^{N-1}$

$[m_1 + Mask..M_1 + Mask]$ if $m_1, M_1 \geq 2^{N-1}$

$[m_1..2^{N-1} - 1] \cup [2^{N-1} + Mask..M_1 + Mask]$ otherwise

Unions of intervals are required here

Dedicated Is/C propagators (2)

For bit-wise operations : very approximated propagation

- A & B = R : propagated like $A \geq R \wedge B \geq R$
- we rely on BV-propagators for these constraints

Simplification rules

- translation into cheap integer arithmetic when possible $(A \oplus B = R \text{ and } A \leq R) \hookrightarrow A + B = R$
- goal = benefits from COLIBRI global reasoning on arith. operators

Congruence propagation

- only parity propagation
- BL and integer translation allow more C-propagation

BV domain

BV : simple abstract domain designed to be combined with Is/C

BV(A) records the known bits of A

- fixed size arrays of values in $\{\bot, 0, 1, \star\}$ (called $\star$-bits)
- $bv_A[k] = 0$ implies that $A[k] = 0$
- $bv_A[k] = 1$ implies that $A[k] = 1$
- $bv_A[k] = \star$ does not imply anything
- $bv_A[k] = \bot$ indicates a contradiction

BV propagators

Propagators : forward and backward propagation of $\star$-bits

Propagators for non-arithmetic operators

- precise and efficient propagation
- especially when one operand is know (mix well with labelling)

Propagators for arithmetic operators

- limited form of bit-blasting inside the propagator
- very restricted propagation
- we rely on Is/C propagators for these constraints

Communication between Is/C and BL

Consistency propagators : designed to enforce consistency between the different domains of a same variable

From BL to Is/C

- if $bl_X = \star 1 \star 101$ then $X \in [21..61]$
- if $bl_X = \star 1 \star 101$ then $X \equiv 5 \ mod \ 8$

From Is/C to BL

- (N=6) if $D_x = [0..15]$ then $bl_X = 00 \star\star\star\star$
- (N=6) if $X \equiv 5 \ mod \ 8$ then $bl_X = \star\star\star 101$

Experiments

Goal : comparison of CP(BV), CP(FD) and SAT

CP(BV) vs CP(FD)

- evaluate the improvement of each new feature
- stability w.r.t. search heuristics

CP(BV) vs CP(FD) vs SAT

- evaluate the current gap between technologies

CP(BV) vs SAT, scalability

- scalability w.r.t. bit-width

Experiments (2)

Test bench

- 164 problems from the SMTLIB or generated by Osmose
- mostly 32-bit, up to 1,700 variables and 17,000 operators
- for scalability : 87 linear and non-linear problems automatically extended to bit-width of 64, 128, 256 and 512

Competitors

- Eclipse/IC
- STP (win. 06), Boolector (win. 08), MathSat (win. 09)

Experiment 1 : CP(BV) vs CP(FD) vs SAT

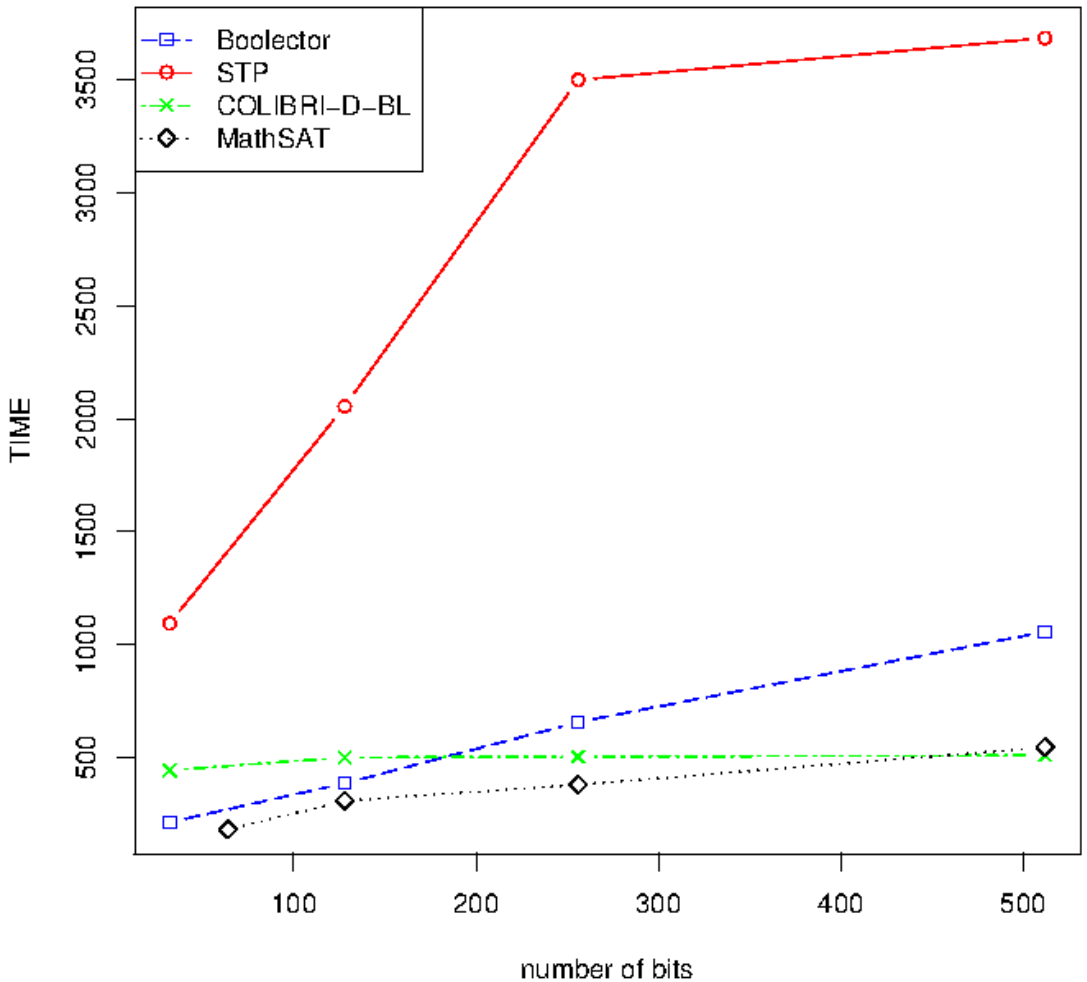
Tool	Category	Time	# success
Eclipse/IC	CP(FD)	1750	79/164
COLIBRI	CP(FD)	2436	43/164
COLIBRI-2009	CP(FD)	1520	89/164
COL-D-BL	CP(BV)	712	138/164
MathSat	SAT	794	128/164
STP	SAT	618	144/164
Boolector	SAT	291	157/164

Time out = 20s

Experiment 2 : CP(BV) vs CP(FD)

Tool	Category	Time	# success
Eclipse/IC-min	CP(FD)	1760	78/164
Eclipse/IC-rand	CP(FD)	2040	72/164
Eclipse/IC-split	CP(FD)	1750	79/164
COL-min	CP(FD)	2436	43/164
COL-rand	CP(FD)	2560	36/164
COL-split	CP(FD)	2550	40 /164
COL-smart	CP(FD)	2475	40/164
COL-2009-min	CP(FD)	1520	89/164
COL-2009-rand	CP(FD)	1513	89/164
COL-2009-split	CP(FD)	1682	85/164
COL-2009-smart	CP(FD)	1410	95/164
COL-D-min	CP(BV)	1453	94/164
COL-D-rand	CP(BV)	1392	96/164
COL-D-split	CP(BV)	1593	89/164
COL-D-smart	CP(BV)	893	125 /164
COL-D-BL-min	CP(BV)	1174	108/164

Experiment 3 : CP(BV) vs SAT, scalability



□ : Boolector
○ : STP
× COL-D-BL
◇ : MathSat

Results

CP(BV) largely outperform CP(FD)

- Results stable w.r.t. the search heuristics
- each feature brings something

Fill the gap with SAT approaches

- yet, the very best SAT approaches still ahead

CP(BV) scales better than SAT w.r.t. bitwidth

- rmk : most scaled examples are UNSAT, and MathSat scales much better on UNSAT formulas

Outline

COLIBRI

CP and bitvectors

CP and arrays

About CP and verification

Overview

Goal : an efficient CP(FD) approach for array+FD constraints

- go beyond standard filtering-based techniques (ELEMENT)
- motivation = software verification

Approach : combine global symbolic deduction mechanisms with local filtering in order to achieve better deductive power than both technique taken in isolation

Results :

- an original "greybox" combination for array+FD constraints
  - identify which information should be shared
  - propose ways of taming communication cost
- a prototype and encouraging experiments (random instances)
  - greater solving power (beats perfect blackbox combination)
  - low overhead
- easy to adapt for any CP solver (small API)

Why a dedicated combination framework ?

Or : more direct approaches, and why we do not choose them

Standard combination scheme between arrays and CP(FD)

[Nelson-Oppen (NO)]

- NO is heavy on non-convex theories like arrays or integers
- FD constraints do not fit well into NO assumptions
  [infinite model]

Remove all *store* functions by introducing $\vee$

- CP not well-adapted for handling case-splits

Simple concurrent black-box combination [first success wins]

- we want to outperform it in solving power
- while still allowing easy re-use of any CP engine

The theory of arrays

The standard theory of arrays is defined by

- three sorts : arrays A, elements of arrays E, indexes I
- function $\mathit{select}(T, i) : A \times I \mapsto E$
- function $\mathit{store}(T, i, e) : A \times I \times E \mapsto A$
- $=$ and $\neq$ over E and I

Semantics (read-over-write)

- (FC) $i = j \longrightarrow \mathit{select}(T, i) = \mathit{select}(T, j)$
- (RoW-1) $i = j \longrightarrow \mathit{select}(\mathit{store}(T, i, e), j) = e$
- (RoW-2) $i \neq j \longrightarrow \mathit{select}(\mathit{store}(T, i, e), j) = \mathit{select}(T, j)$

CP and arrays : local filtering

- arrays represented by pairs (*index*, *element*)
  [explicit arrays of logical variables]
- constraints on domains of indexes / elements (and size)
- *select* : well-known constraint ELEMENT
  [Van Hentenryck-Carillon 88, Brand 01]
- *store* : more recent work [Charreteur-Botella-Gotlieb 09]

```
Element(ARRAY,I,E) :-
(
  integer(I)? ARRAY[I] == E, success
;
  D(E) ← D(E) ∩ ⋃_{i∈D(I)} D(ARRAY(i)),
  D(I) ← {i ∈ D(I) | D(E) ∩ D(ARRAY[i]) ≠ ∅},
  wait(...)
)
```

CP and arrays : local filtering

- arrays represented by pairs (*index*, *element*)
  [explicit arrays of logical variables]
- constraints on domains of indexes / elements (and size)
- *select* : well-known constraint ELEMENT
  [Van Hentenryck-Carillon 88, Brand 01]
- *store* : more recent work [Charreteur-Botella-Gotlieb 09]

```
Update(A,I,E,A') :-
(
  integer(I)? A'[I]==E, ∀ k ≠ I do A'[k]==A[k], success
;
  D(E) ← D(E) ∩ ⋃_{i∈D(I)} D(A'(i)),
  D(I) ← {i ∈ D(I) | D(E) ∩ D(A'[i]) ≠ ∅},
  ∀ k ∉ D(I) do A'[k] == A[k]
  ∀ k ∈ D(I) do D(A'[k]) ← D(A'[k]) ∩ (D(A[k]) ∪ D(E))
  ... )
```

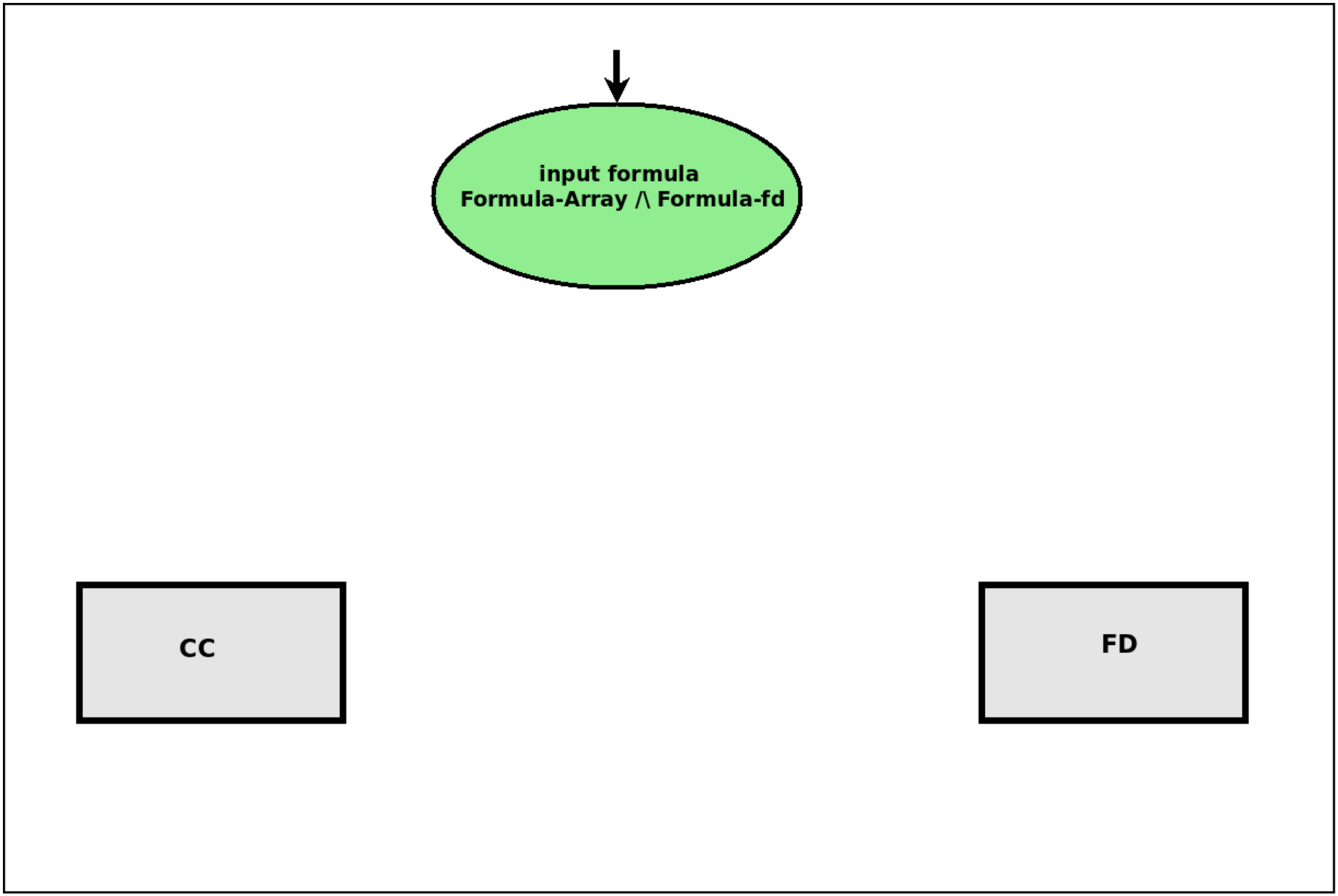
CP and arrays : local filtering

- arrays represented by pairs (*index*, *element*)
  [explicit arrays of logical variables]
- constraints on domains of indexes / elements (and size)
- *select* : well-known constraint ELEMENT
  [Van Hentenryck-Carillon 88, Brand 01]
- *store* : more recent work [Charreteur-Botella-Gotlieb 09]

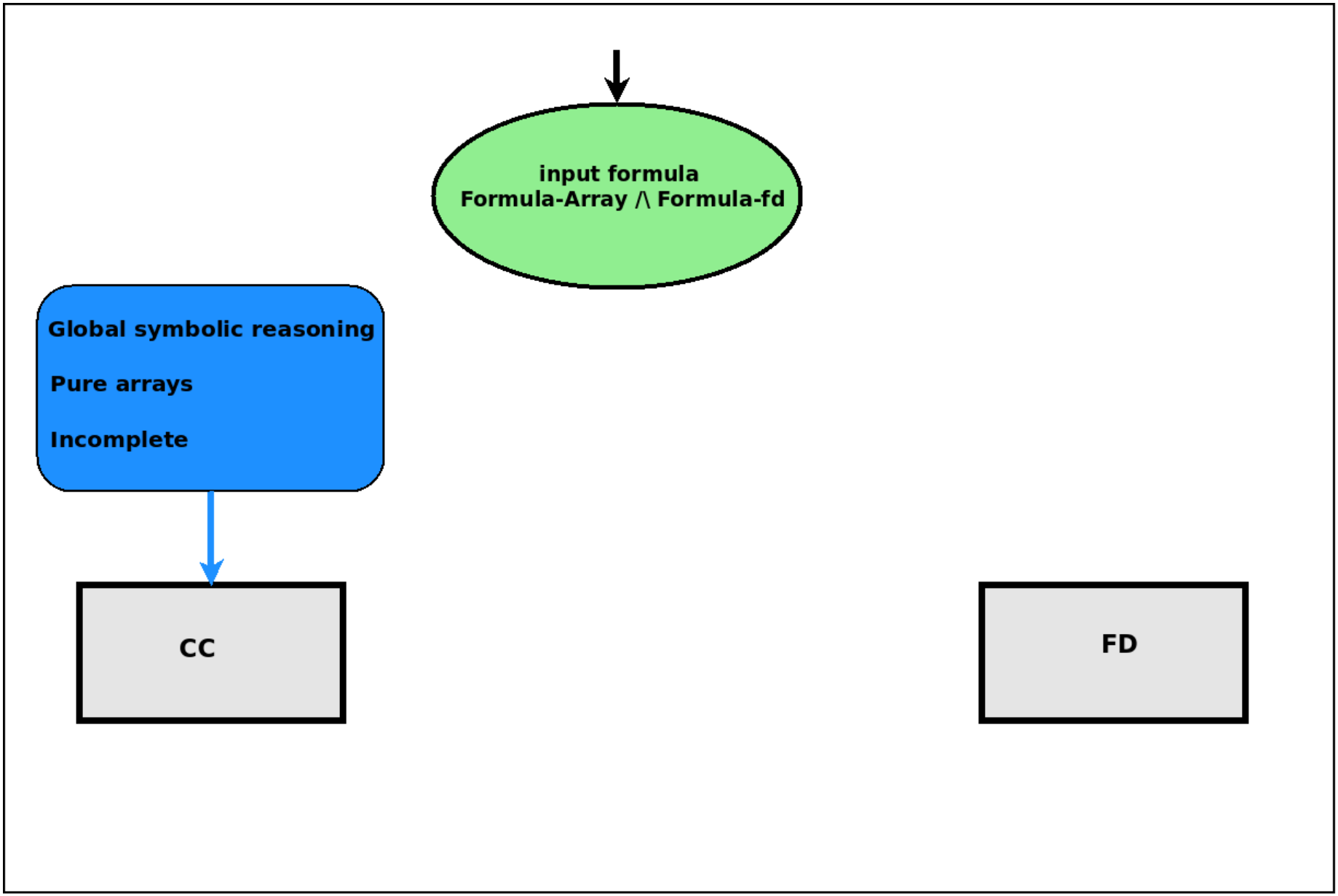
Insufficient for many array constraints from program verification

- long chains of updates, variable indexes
  [see formulas from SMT-LIB]
- symbolic reasoning required

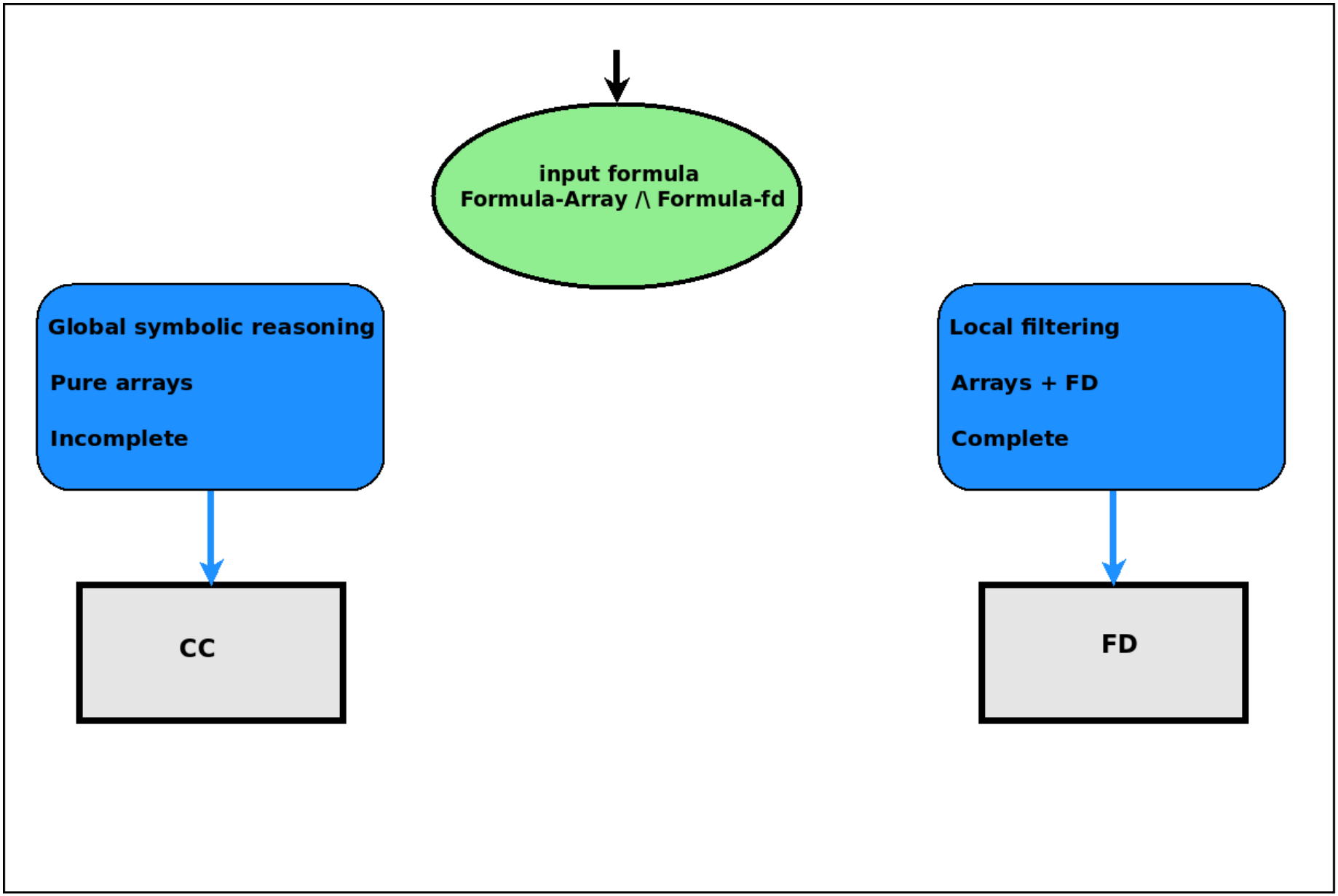
Our approach



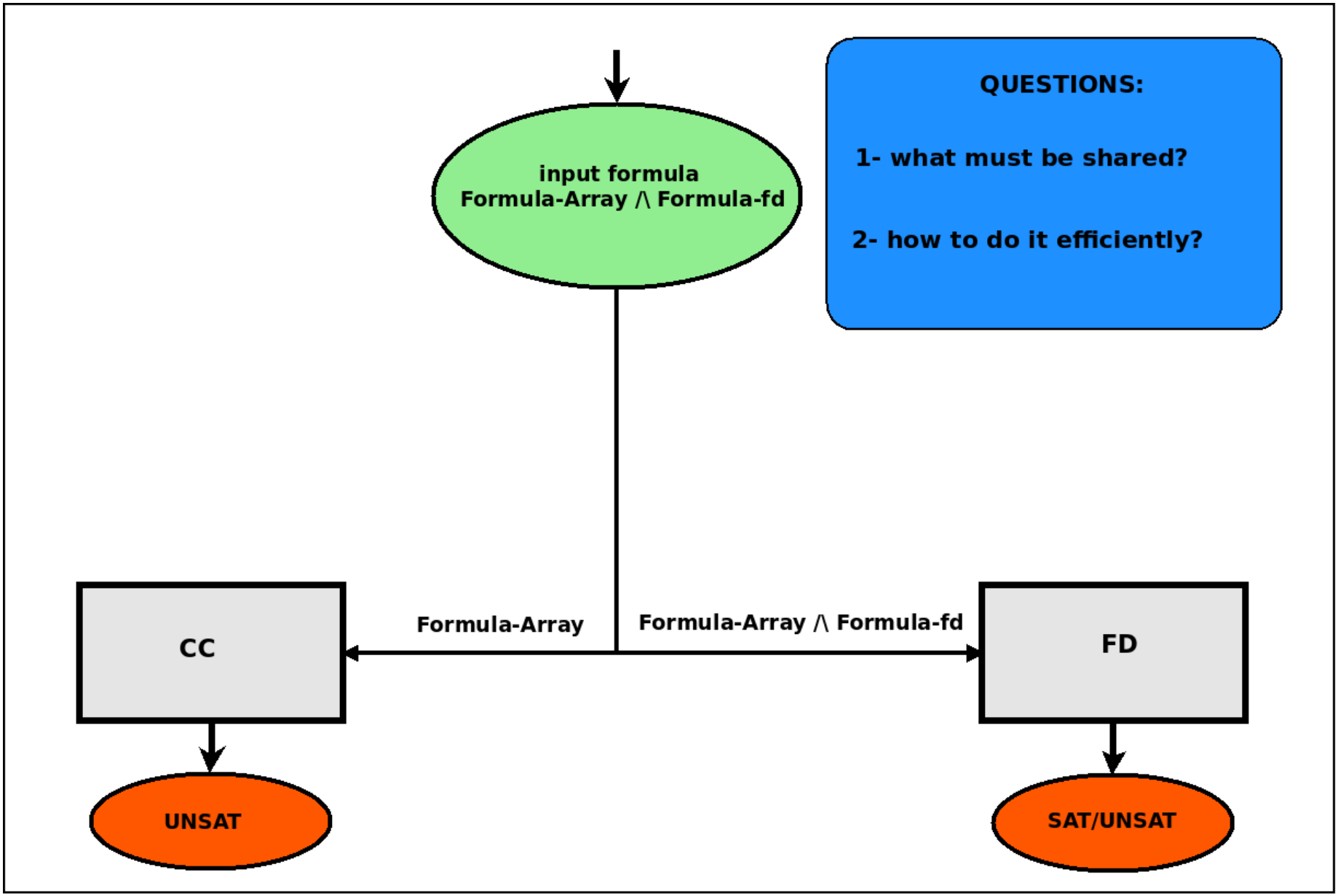
Our approach



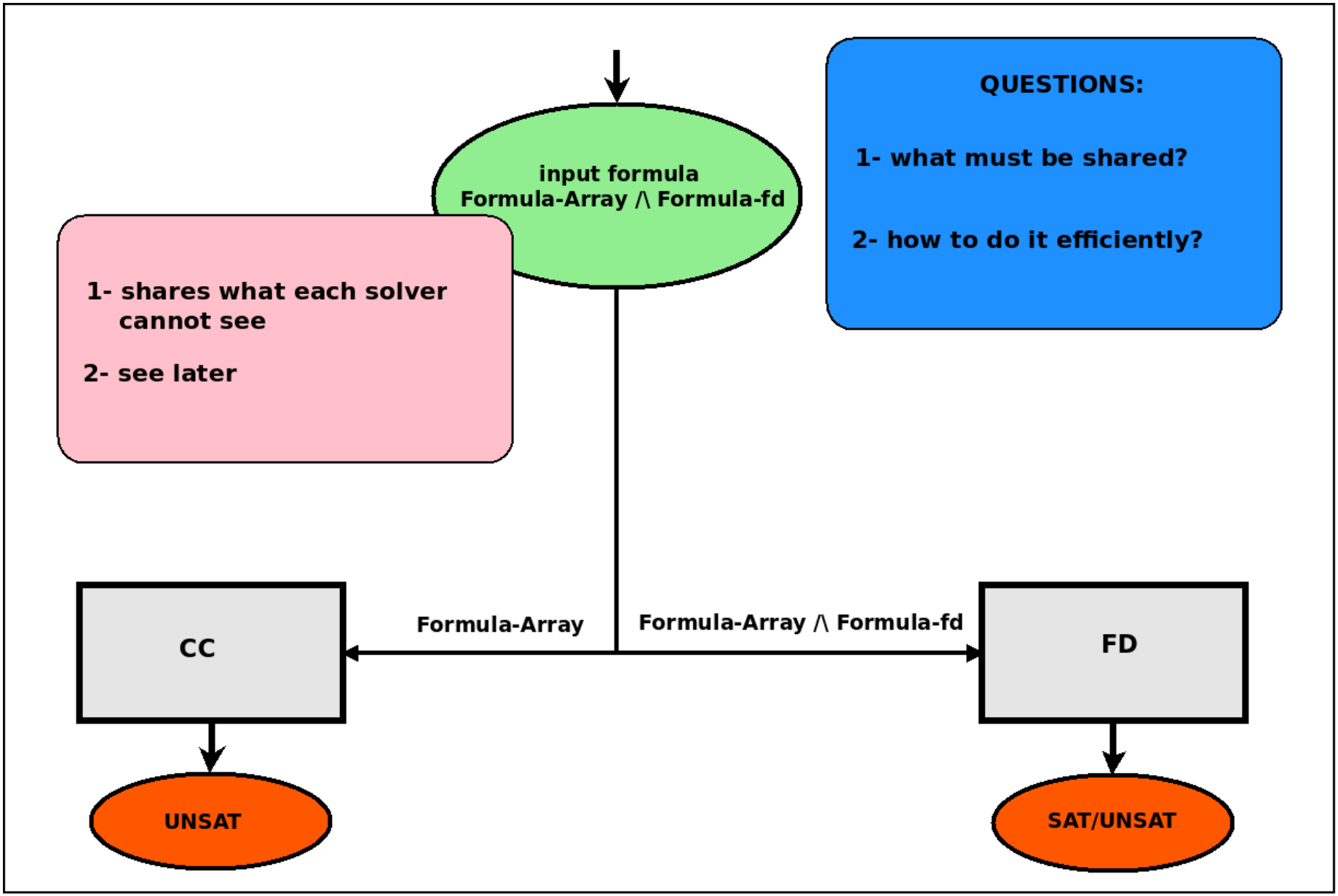
Our approach



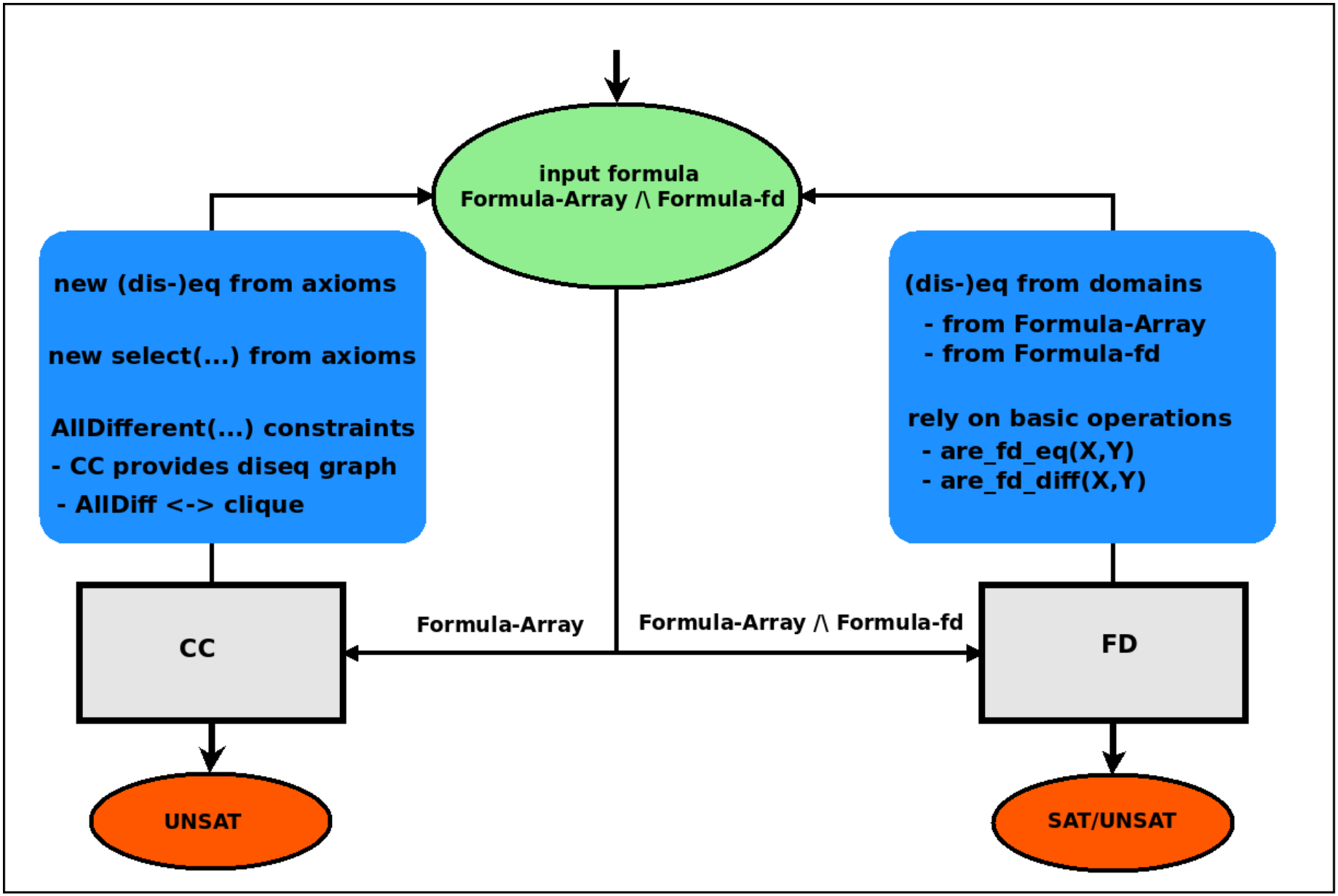
Our approach



Our approach



Our approach



Examples

$$e = \mathit{select}(T, i) \wedge f = \mathit{select}(T, j) \wedge e \neq f \wedge i = j$$

T array of size 100
Domains : 0..100

- ✓ CC : *unsat* quickly (axiom FC)
- × FD : needs labelling [no answer in 60 min in COMET]
- ✓ FDCC : *unsat* quickly through CC

Examples

$$e = \mathit{select}(T, i) \wedge f = \mathit{select}(T, j) \wedge e \neq f \wedge i = j$$

T array of size 100
Domains : 0..100

✓ CC : *unsat* quickly (axiom FC)

✗ FD : needs labelling [no answer in 60 min in COMET]

✓ FDCC : *unsat* quickly through CC

Examples

$$e = \mathit{select}(T, i) \wedge f = \mathit{select}(T, j) \wedge e \neq f \wedge i = j$$

T array of size 100
Domains : 0..100

✓ CC : *unsat* quickly (axiom FC)

✗ FD : needs labelling [no answer in 60 min in COMET]

✓ FDCC : *unsat* quickly through CC

Examples

$$e = \mathit{select}(T, i) \wedge f = \mathit{select}(T, j) \wedge e \neq f \wedge i = j$$

T array of size 100
Domains : 0..100

- ✓ CC : *unsat* quickly (axiom FC)
- ✗ FD : needs labelling [no answer in 60 min in COMET]
- ✓ FDCC : *unsat* quickly through CC

Examples

$$i \in 1..5 \wedge j \in 6..10 \wedge a \neq select(store(store(T, j, a), i, b), j)$$

- ✗ CC : no deduction since $i \neq j$ cannot be inferred
- ✗ FD : needs labelling, cannot established $select(store(T, j, a), j) = a$
- ✓ FDCC : FD deduces $i \neq j$ (domain-check), CC can then deduce $a \neq select(store(T, j, a), j)$ then $a \neq a$ and *unsat*

Examples

$$i \in 1..5 \wedge j \in 6..10 \wedge a \neq select(store(store(T, j, a), i, b), j)$$

- ✗ CC : no deduction since $i \neq j$ cannot be inferred
- ✗ FD : needs labelling, cannot established $select(store(T, j, a), j) = a$
- ✓ FDCC : FD deduces $i \neq j$ (domain-check), CC can then deduce $a \neq select(store(T, j, a), j)$ then $a \neq a$ and *unsat*

Examples

$$i \in 1..5 \wedge j \in 6..10 \wedge a \neq select(store(store(T, j, a), i, b), j)$$

- ✗ CC : no deduction since $i \neq j$ cannot be inferred
- ✗ FD : needs labelling, cannot established $select(store(T, j, a), j) = a$
- ✓ FDCC : FD deduces $i \neq j$ (domain-check), CC can then deduce $a \neq select(store(T, j, a), j)$ then $a \neq a$ and *unsat*

Examples

$$i \in 1..5 \wedge j \in 6..10 \wedge a \neq \mathit{select}(\mathit{store}(\mathit{store}(T, j, a), i, b), j)$$

- ✗ CC : no deduction since $i \neq j$ cannot be inferred
- ✗ FD : needs labelling, cannot established $\mathit{select}(\mathit{store}(T, j, a), j) = a$
- ✓ FDCC : FD deduces $i \neq j$ (domain-check), CC can then deduce $a \neq \mathit{select}(\mathit{store}(T, j, a), j)$ then $a \neq a$ and *unsat*

Examples

$$e = select(T, i) \wedge f = select(T, j) \wedge g = select(T, k)$$
$$\wedge e \neq f \wedge e \neq g \wedge f \neq g$$

T array of size 2, domain of indexes 1..2

- ✗ CC : deduces ALLDIFFERENT(i,j,k), does not output *unsat* (domains not taken into account)
- ✗ FD : needs labelling [labels indexes first ! !]
- ✓ FDCC : CC deduces ALLDIFFERENT(i,j,k), then FD deduces *unsat*

Examples

$$e = \mathit{select}(T, i) \wedge f = \mathit{select}(T, j) \wedge g = \mathit{select}(T, k)$$
$$\wedge e \neq f \wedge e \neq g \wedge f \neq g$$

T array of size 2, domain of indexes 1..2

- ✗ CC : deduces ALLDIFFERENT(i,j,k), does not output *unsat* (domains not taken into account)
- ✗ FD : needs labelling [labels indexes first ! !]
- ✓ FDCC : CC deduces ALLDIFFERENT(i,j,k), then FD deduces *unsat*

Communication framework

Communication between FD and CC can be costly

- especially, checking (dis-)equalitites of variables through their domains, $|V|^2$ pairs to be checked

How to tame communication costs ?

- a **communication policy** allowing tight control over expensive communications
- a reduction of the number of pairs of variables to consider (**critical pairs**)

Other

- labelling is only transmitted to FD

Communication framework (2)

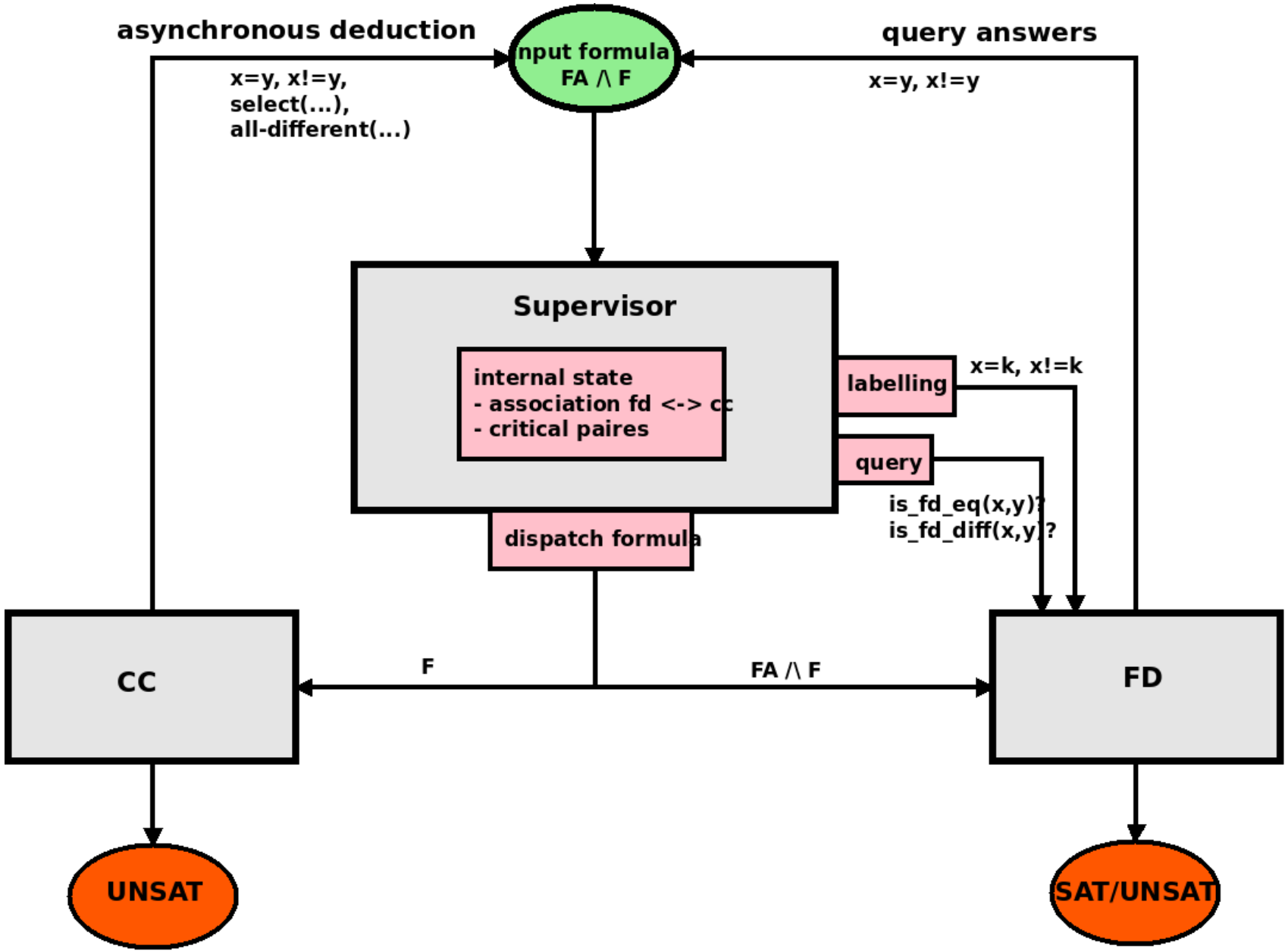
Communication policy

- cheap communications ($\mathrm{CC} \mapsto \mathrm{FD}$) made asynchronously
- expensive ones ($\mathrm{FD} \mapsto \mathrm{CC}$) made on request (supervisor)

Critical pairs

- focus on pairs whose (dis-)equality will surely lead to new deductions in CC [see axioms]
- focus on **critical pairs**, involved in RoW-* rules
  - e.g. for $v \hat{=} \mathit{select}(\mathit{store}(T, i, e), j)$: pairs (i, j) and (e, v)
- linear in $\#\mathit{select}$, capture the specificity of array axioms
- manageable in practise, still brings interesting deductions
- incremental computation of critical paires by CC

Communication framework (3)



Experiments

Random formulas

- four kind of formulas (easy / hard array, arith / no arith)
- length 10-60

Properties to be evaluated

- ability to solve as many formulas as possible
- comparison with FD and CC [including overhead]
- comparison with blackbox combinations (HYBRID and BEST)

Experiment 1 : evaluates on 369 formulas, balanced in the 4 classes and *sat* / *unsat*

Experiment 2 : evaluates on 100 formulas for each length between 10 and 60 [performance w.r.t. complexity threshold]

Results

First experiment

- solving power : solves $>$ than FD, CC, or BEST
  - 22 formulas (out of 369) solved only by FDCC
  - 5x less TO than FD and 3x less TO than BEST
- affordable overhead over CC and FD [when they succeed]
  - at worst 4x slower, on average 1.1x - 1.5x slower
- robustness : results hold for all 4 classes and *sat* / *unsat*

Second experiment

- FDCC again better than FD and CC
- maximal benefits on hard-to-solve formulas
  [closed to complexity threshold]

Results (2)

	Total (369)			
	S	U	TO	T
CC	29	115	225	13545
FD	154	151	64	3995
FDCC	**181**	175	**13**	**957**
BEST	154	175	40	2492
HYBRID	154	175	40	2609

S : # sat answer, U : # unsat answer,
TO : # time-out (60 sec), T : time in sec.

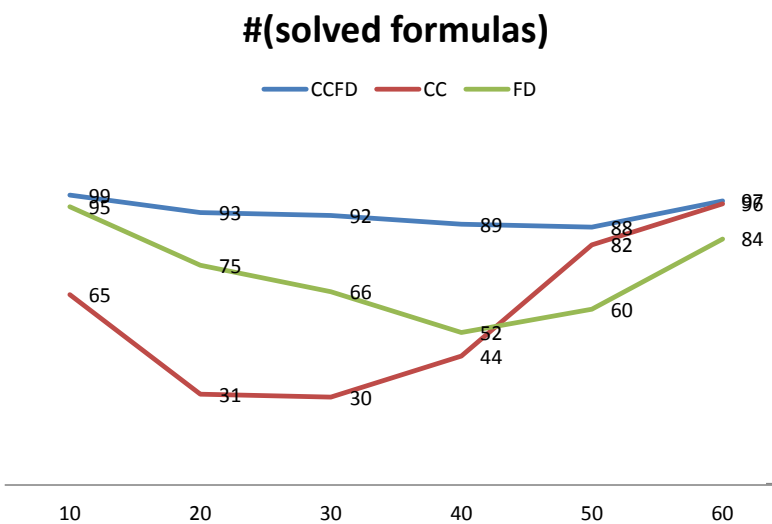
Results (2)

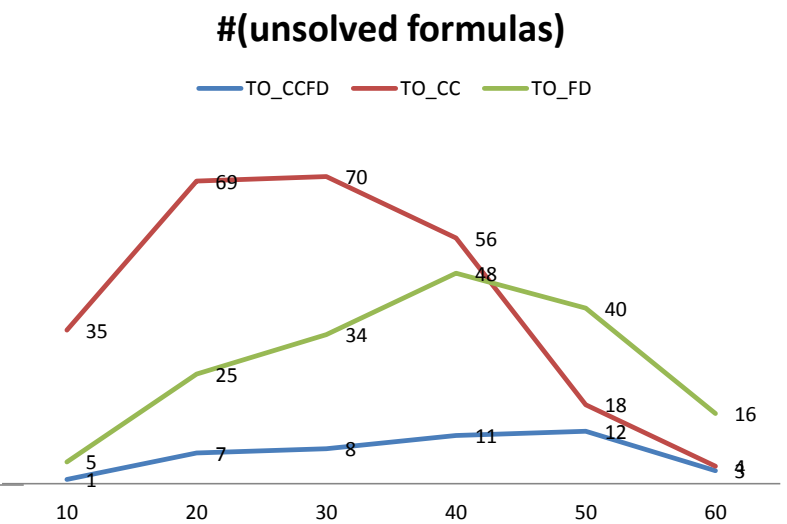
	AEUF-I (79)				AEUF-II (90)			
	S	U	TO	T	S	U	TO	T
CC	26	37	16	987	2	30	58	3485
FD	39	26	14	875	35	18	37	2299
FDCC	**40**	37	**2**	**144**	**51**	30	**9**	**635**
BEST	39	37	3	202	35	30	25	1529
HYBRID	39	37	3	242	35	30	25	1561

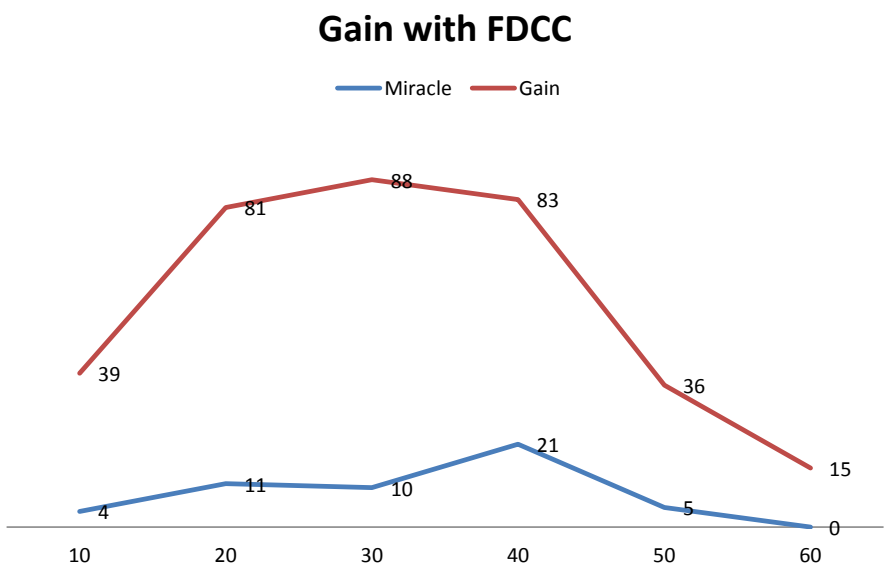
	AEUF+LIA-I (100)				AEUF+LIA-II (100)			
	S	U	TO	T	S	U	TO	T
CC	1	21	78	4689	0	27	73	4384
FD	50	47	3	199	30	60	10	622
FDCC	**52**	48	**0**	**24**	**38**	60	**2**	**154**
BEST	50	48	2	139	30	60	10	622
HYBRID	50	48	2	159	30	60	10	647

S : # sat answer, U : # unsat answer,
TO : # time-out (60 sec), T : time in sec.

Results (2)


#(solved formulas)
CCFD
CC
FD
99
95
93
92
89
88
82
97
96
84
75
66
60
65
31
30
52
44
10
20
30
40
50
60


#(unsolved formulas)
TO_CCFD
TO_CC
TO_FD
69
70
56
48
40
35
25
34
16
18
12
5
7
8
11
1
3
4
10
20
30
40
50
60


Gain with FDCC
Miracle
Gain
88
81
83
39
36
15
21
11
10
5
4
0
10
20
30
40
50
60

Conclusion

Results

- an original decision procedure for arrays that combines ideas from symbolic reasoning and finite-domain constraint solving
  - identify which information should be shared
  - propose ways of taming communication cost
- a prototype and encouraging experiments (random instances)
  - greater solving power (beats even BEST)
  - low overhead
- easy to adapt for any CP solver

Future work

- experiments on real-life problems
- extend the approach to handle memory heaps (`new`, `delete`)

Outline

COLIBRI

CP and bitvectors

CP and arrays

About CP and verification

About CP and verification

CP can be tuned for real-size verification problems

- at least for testing
- ✓ rather good at proving sat, and produce a witness !
- × but not very good for valid and $\Rightarrow$

Beware : forget about any propagation based on concrete enumeration (can be hidden in learning, global constraints, etc.)

A few interesting features of CP

- approximation of disjunctive constraints
- dynamic addition of constraints, controlled from the propagators
- easy to quickly support a new constraint : needs only a checker

CP and Abstract Interpretation (AI)

Clearly the same kind of domains and computations

- non-relational domains in AI $\leftrightarrow$ local domain in CP
- relational domain in AI $\leftrightarrow$ global constraint in CP

CP borrowed the widening and $\sqcup$ approach

- maybe AI could borrow the domain-split trick / lazy unrolling

CP & SMT

Clearly, complementary strengths

Two very different ways of exploring a complex formula

- SMT : learning from previous mistakes (past of exploration)
- CP : overapproximation of disjunctions (future of exploration)

About enumeration

- CP : $D^{|V|}$ valuations
- SMT, convex theories : $2^{|atoms|}$ queries
- SMT, non-convex theories : $2^{|atoms|+|V|^2}$ queries