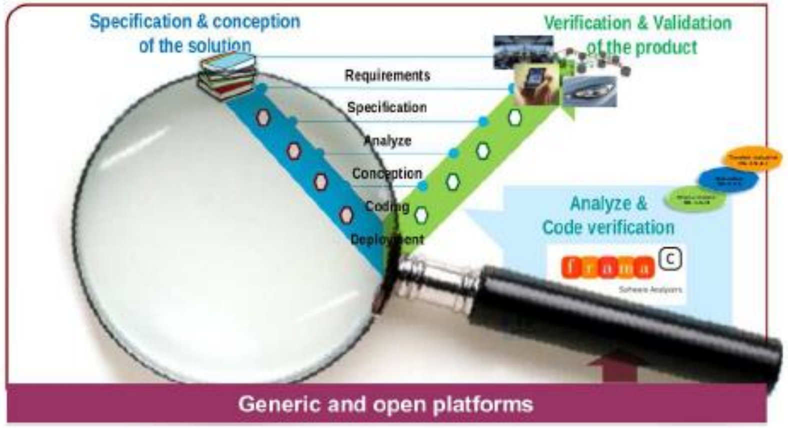
SOFTWARE VERIFICATION:
PAST, PRESENT & FUTURE
(about verif, constraints and learning)

Sébastien Bardin (CEA LIST)

ABOUT MY LAB @CEA

CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- second part of V-cycle
- automatic software analysis
- mostly source code



frama C Software Analyzers

GATeL

PathCrawler

UNISIM

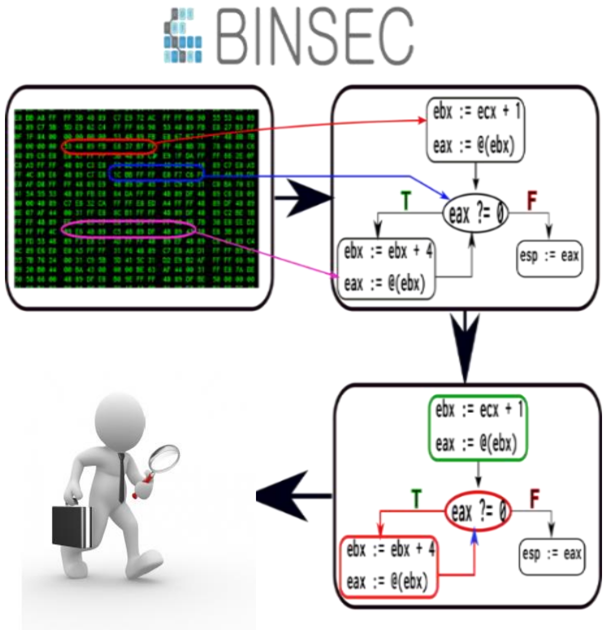
POLAR SSL

ME, MYSELF and I

- **Interested in designing methods & tools helping to develop very safe/secure systems**

- **Technical core**
  - Formal methods
  - Logic and automated reasoning

- **Application fields**
  - Software engineering
  - Security



THE TALK IN A NUTSHELL

- **Software Verification is highly successful**
  - Change the game in highly-regulated fields
  - Start spreading to less critical fields

- **Yet, not so much investigated by the AI community**

- **Goal of the talk: initiate discussion**
  - [past] a tour on software verification & formal methods
  - [present] logic-based verification, an opportunity for CP
  - [future] verification & machine learning

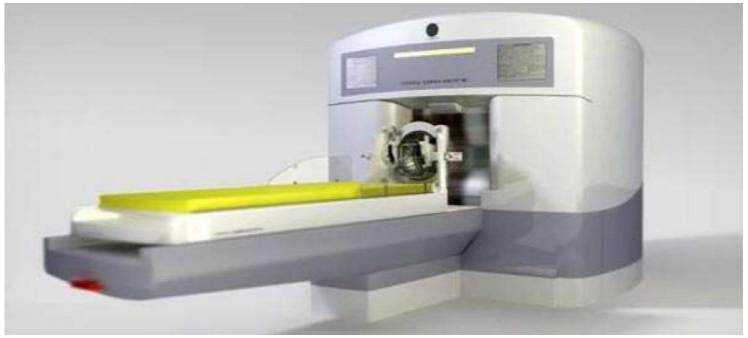
OUTLINE

- Introduction
- [past] a tour on software verification & formal methods
- [present] logic-based verification, an opportunity for CP
- [future] verification & machine learning
- Conclusion

BACK IN TIME: THE SOFTWARE CRISIS (1969)

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful ! To put it quite bluntly : as long as there were no machines, programming was no problem at all ; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

- Edsger Dijkstra, The Humble Programmer (EWD340)



http ://en.wikipedia.org/wiki/List_of_software_bugs

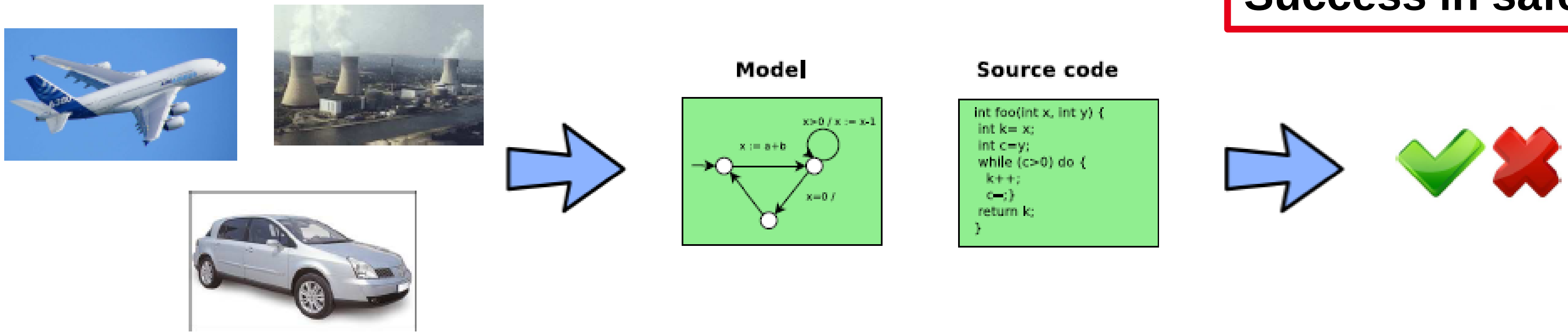
Testing can only reveal the presence of errors but never their absence.

- E. W. Dijkstra (Notes on Structured Programming, 1972)

ABOUT FORMAL METHODS

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Success in safety-critical



```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
  k++;
  c–;}
 return k;
}
```

Key concepts : $M \models \varphi$

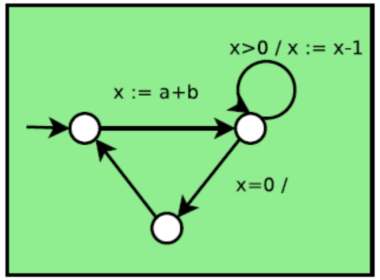
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties

Input model?

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

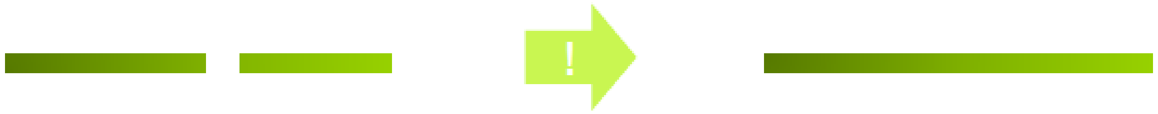
```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

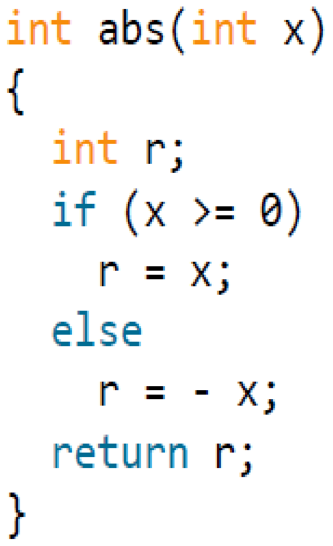
A set of relevant behaviours

- **Reachable states**
- **Traces (finite or infinite)**
- **Execution Tree**
- **…**

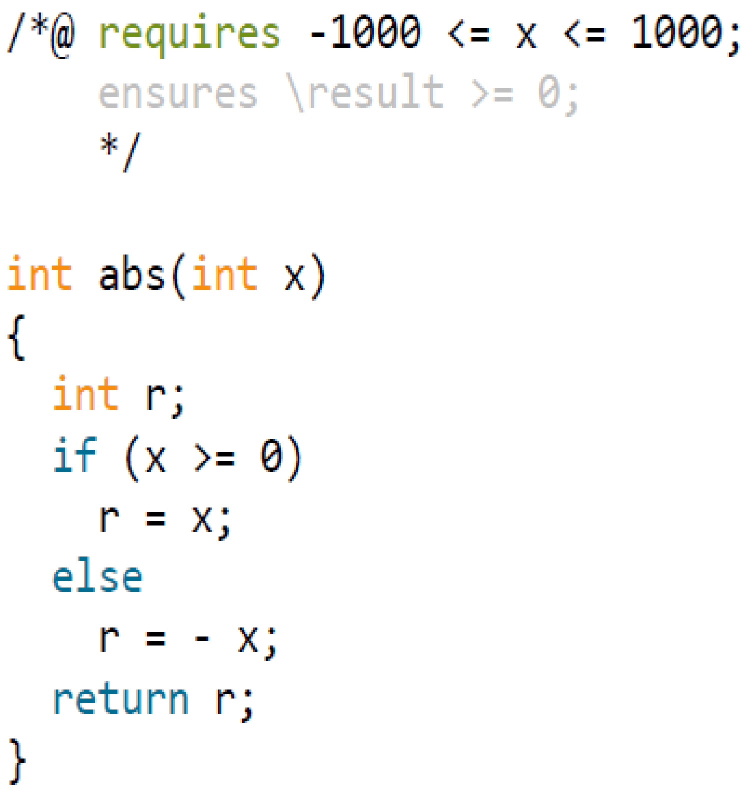
Specification?

Properties are formalized using unequivocal specifications





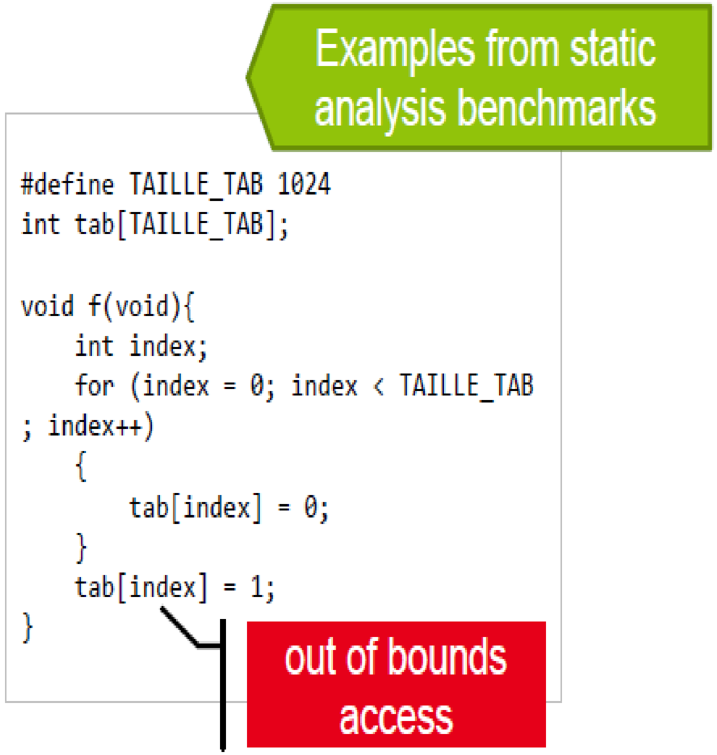
```
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```



```
/*@ requires -1000 <= x <= 1000;
    ensures \result >= 0;
    */

int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

Specification?

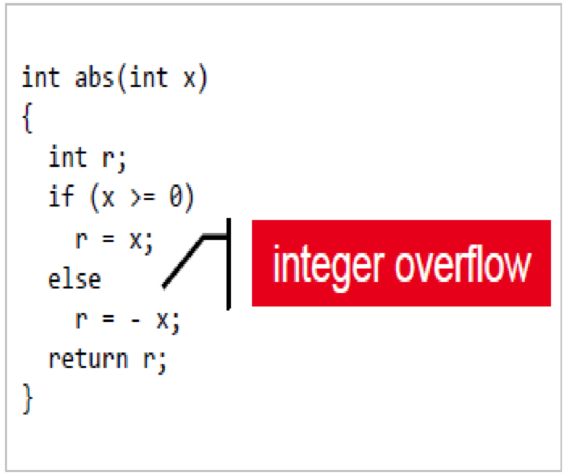
- Code is free of runtime errors
  - `*prt;`
  - `buf[i+1];`
  - `num / den_nz;`
  - `MAX_INT+1;`



Examples from static analysis benchmarks

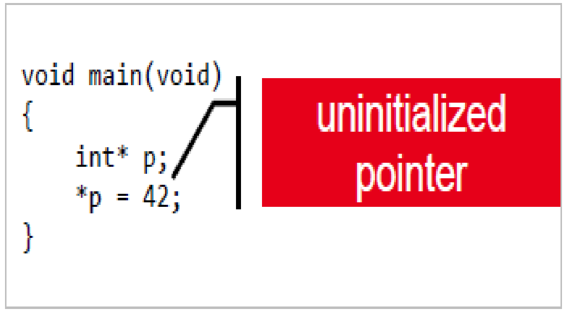
```
#define TAILLE_TAB 1024
int tab[TAILLE_TAB];

void f(void){
    int index;
    for (index = 0; index < TAILLE_TAB
; index++)
    {
        tab[index] = 0;
    }
    tab[index] = 1;
}
```

out of bounds access



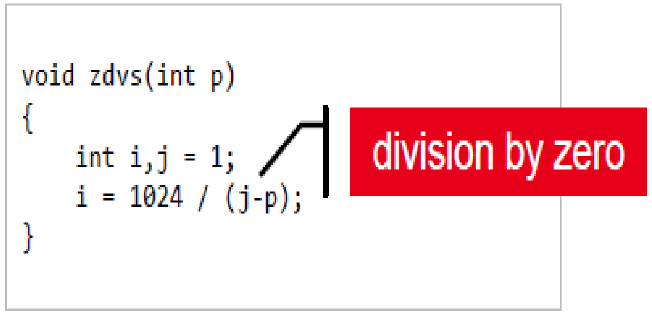
```
int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

integer overflow



```
void main(void)
{
    int* p;
    *p = 42;
}
```

uninitialized pointer



```
void zdvs(int p)
{
    int i,j = 1;
    i = 1024 / (j-p);
}
```

division by zero

Algorithmic check?

Problem is often undecidable
- **Over-appaorximation**
- **Under-approximation**
- **Witness?**

- Abstract Interpretation [1977, Cousot]
- Model checking [1981, Clarke - Sifakis]
- Weakest precondition calculi [197x, Hoare?]

A DREAM COME TRUE … IN CERTAIN DOMAINS

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex : Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C *]
- numerical precision [Fluctuat *]
- source-binary conformance [CompCert]
- ressource usage [Absint]

* : by CEA DILS/LSL

A DREAM COME TRUE … IN CERTAIN DOMAINS (2)

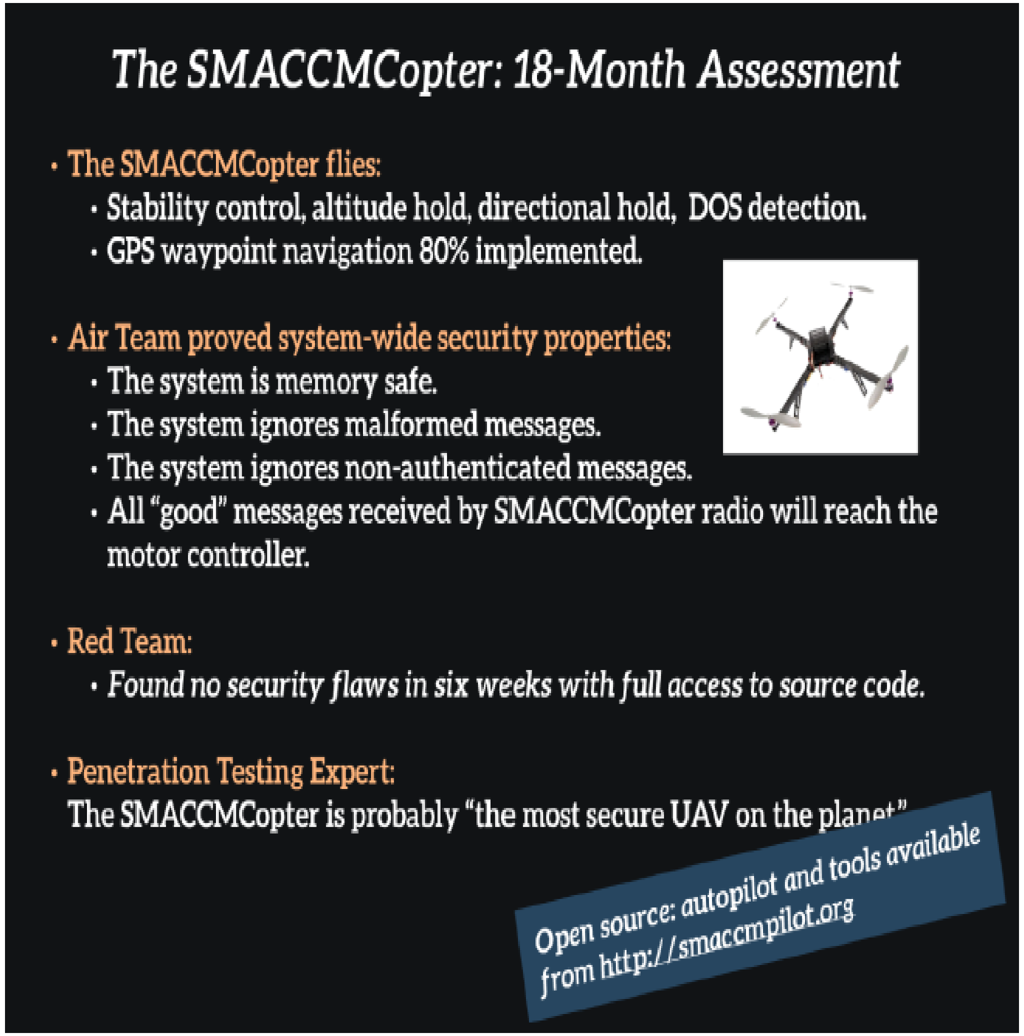
Ex : Microsoft

Verification of drivers [SDV]

- conformance to MS driver policy
- home developers
- and third-party developers

> *Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.*
>
> \- Bill Gates (2002)

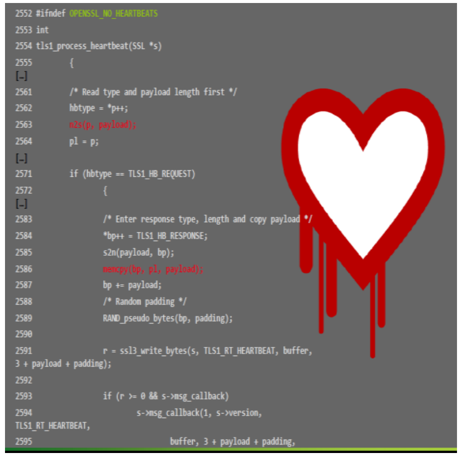
Other successes



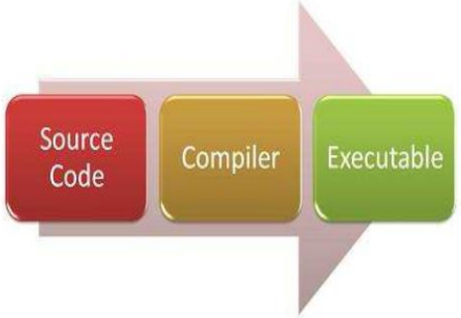
The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.
- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACCMCopter radio will reach the motor controller.
- Red Team:
  - *Found no security flaws in six weeks with full access to source code.*
- Penetration Testing Expert:
  The SMACCMCopter is probably "the most secure UAV on the planet"

Open source: autopilot and tools available from http://smaccmpilot.org



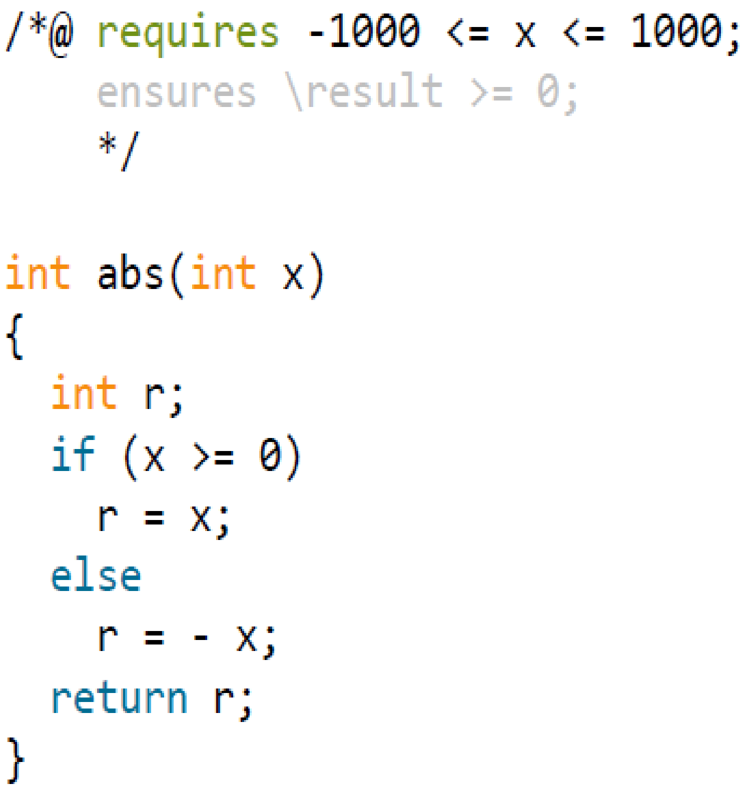
```
#ifndef OPENSSL_NO_HEARTBEATS
int
tls1_process_heartbeat(SSL *s)
        {
[…]
        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;
[…]
        if (hbtype == TLS1_HB_REQUEST)
                {
[…]
                /* Enter response type, length and copy payload */
                *bp++ = TLS1_HB_RESPONSE;
                s2n(payload, bp);
                memcpy(bp, pl, payload);
                bp += payload;
                /* Random padding */
                RAND_pseudo_bytes(bp, padding);

                r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
3 + payload + padding);

                if (r >= 0 && s->msg_callback)
                        s->msg_callback(1, s->version,
TLS1_RT_HEARTBEAT,
                                buffer, 3 + payload + padding,
```



Compcert

SAGE

Weakest precondition



```
/*@ requires -1000 <= x <= 1000;
    ensures \result >= 0;
    */

int abs(int x)
{
  int r;
  if (x >= 0)
    r = x;
  else
    r = - x;
  return r;
}
```

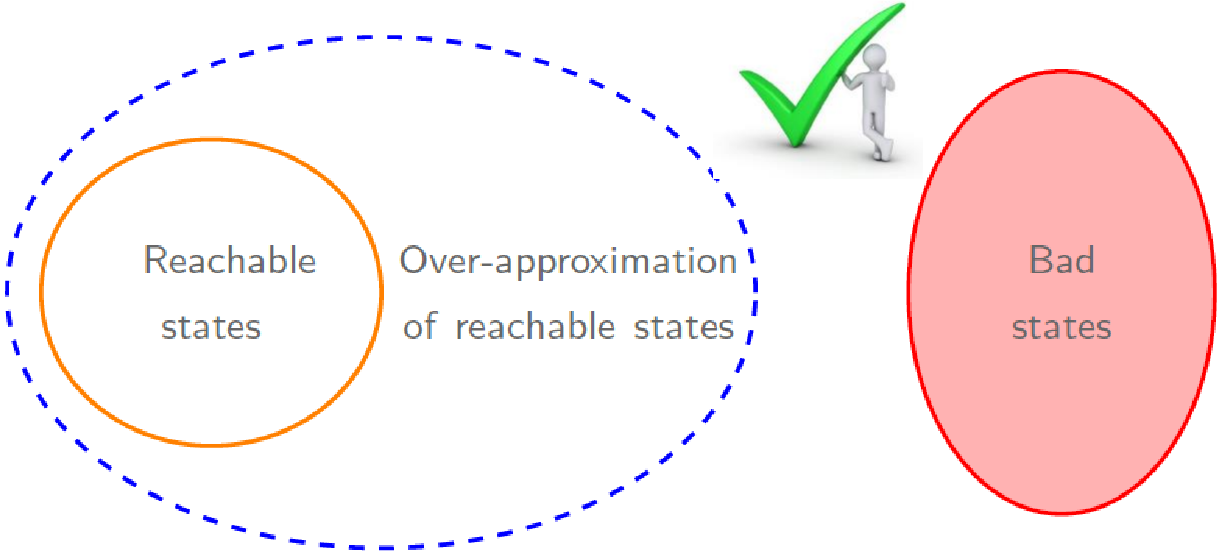
- **Inference of weakest precond wp**
- **Check that Precond => wp**

Powerful & generic but

- Need annotations (loop invariants)
- Need powerful automatic solvers (or by hand)
- Complex properties/semantic: full of quantifiers!

ABSTRACT INTERPRETATION

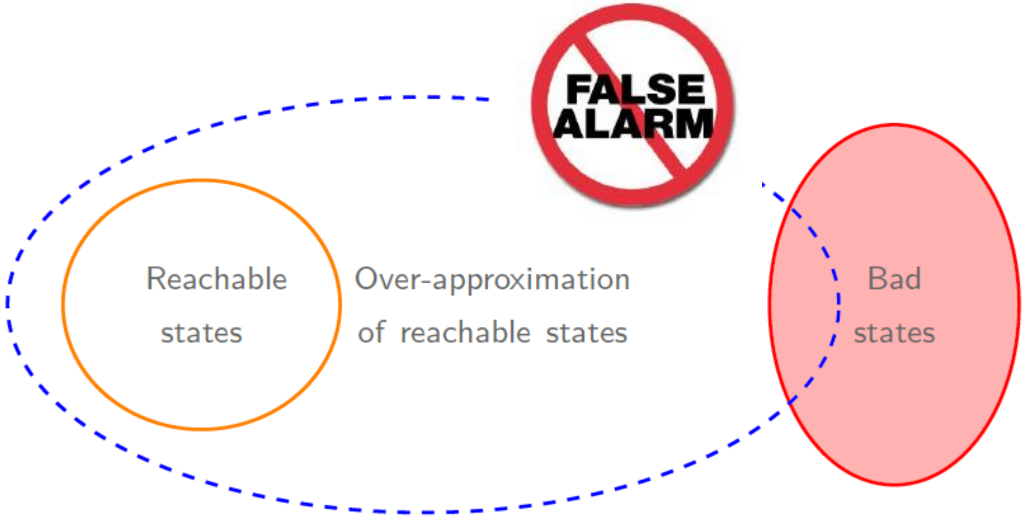
$$(\mathcal{P}(states), \cup, \cap, \rightarrow) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} (states^{\#}, \sqcup, \sqcap, \rightarrow^{\#})$$



Reachable states

Over-approximation of reachable states

Bad states



Reachable states

Over-approximation of reachable states

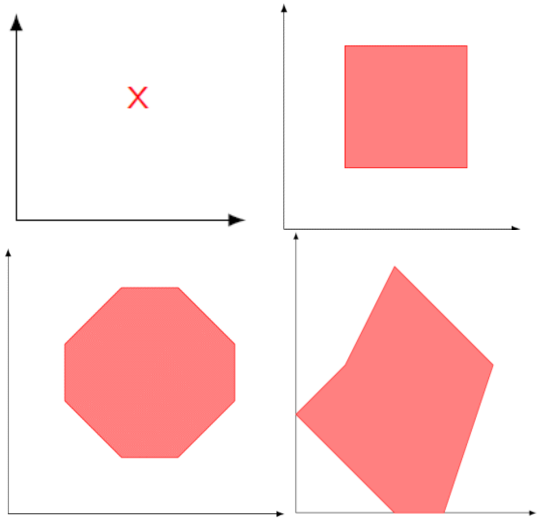
Bad states

ABSTRACT INTERPRETATION (2)

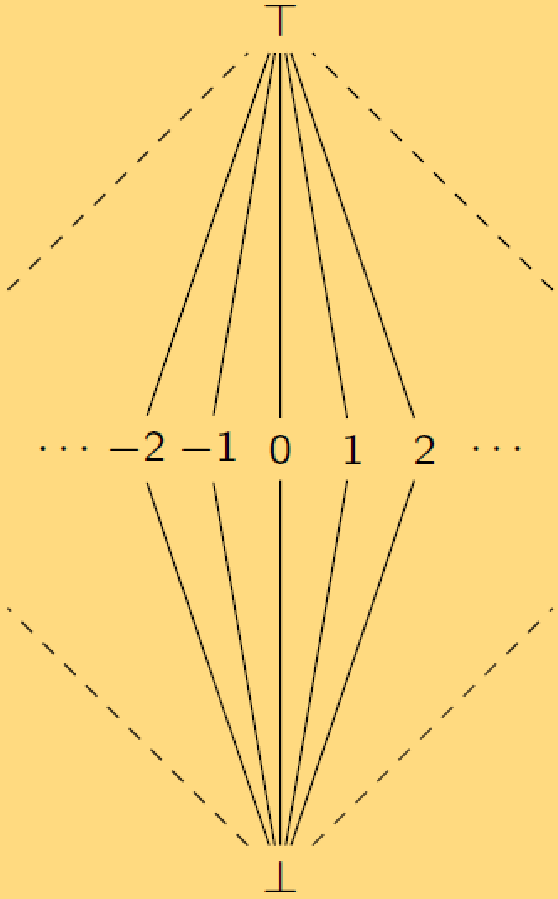
Framework : abstract interpretation

- notion of abstract domain

  $\bot, \top, \sqcup, \sqcap, \sqsubseteq, \text{eval}^{\#}$
- more or less precise domains
  - intervals, polyhedra, etc.
- fixpoint until stabilization



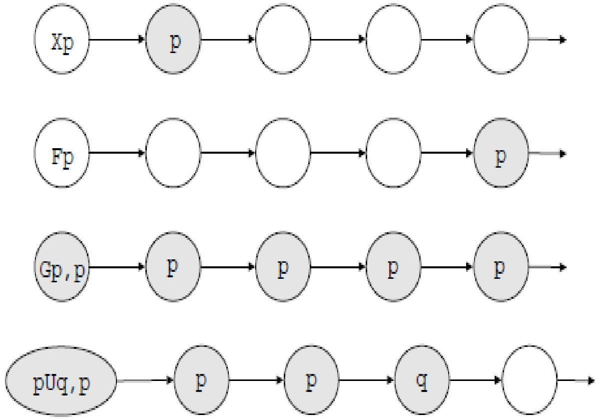
Generalize constant propagation



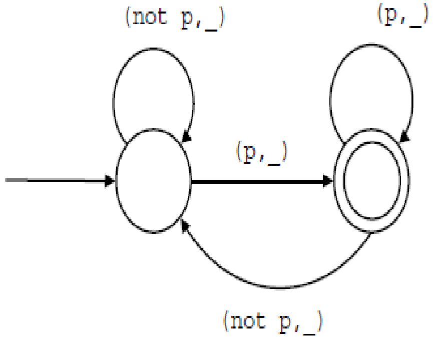
Model checking

(finite) Reactive systems

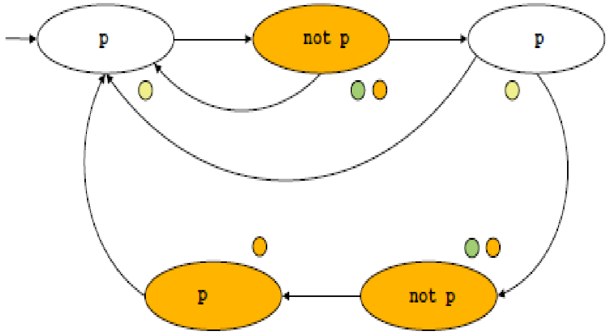
spec = temporal logics



Check: graph & automata algos



Exemple de AGF*p*



OUTLINE

- **Introduction**
- **[past] a tour on software verification & formal methods**
- **[present] logic-based verification, an opportunity for CP**
- **[future] verification & machine learning**
- **Conclusion**

The era of logic-based verification

Current big trend of verification: « reduction to logic »

- **And let the automatic solver do its job**

Fruitful

- Bug finding:
  - Bounded model checking
  - Symbolic execution
- Proof:
  - k-induction
  - Interpolation, pdr

Current state

- **WP: logic**
- **MC: logic**
- **Ab.I: domains [+ logic]**

Why now?

Before 2000's: thm provers adapted to maths, not verif

- **No theory support (integers, arrays, etc.)**
- **No model generation**
- **Bad on complex boolean parts**
- **// full verif: need fixpoint or quantif + inductive pred.**

- WP: automation++
- MC: loop-free reasoning

And then comes the miracle: SMT solvers

- **Complex boolean part, multi-theories, models**
- **Very elegant ways to extend (developer)**
- **Easy to use (user)**
- **// quantifiers still possible, but not so good**

<aparté> SMT solvers

Usual setting

- Quantifier-free formula
- Multi-theories T1 x T2 x …   // some restriction
- Arbitrary Boolean skeleton
- Goal = answer unsat or give a solution

Key 1: Nelson-Oppen combination

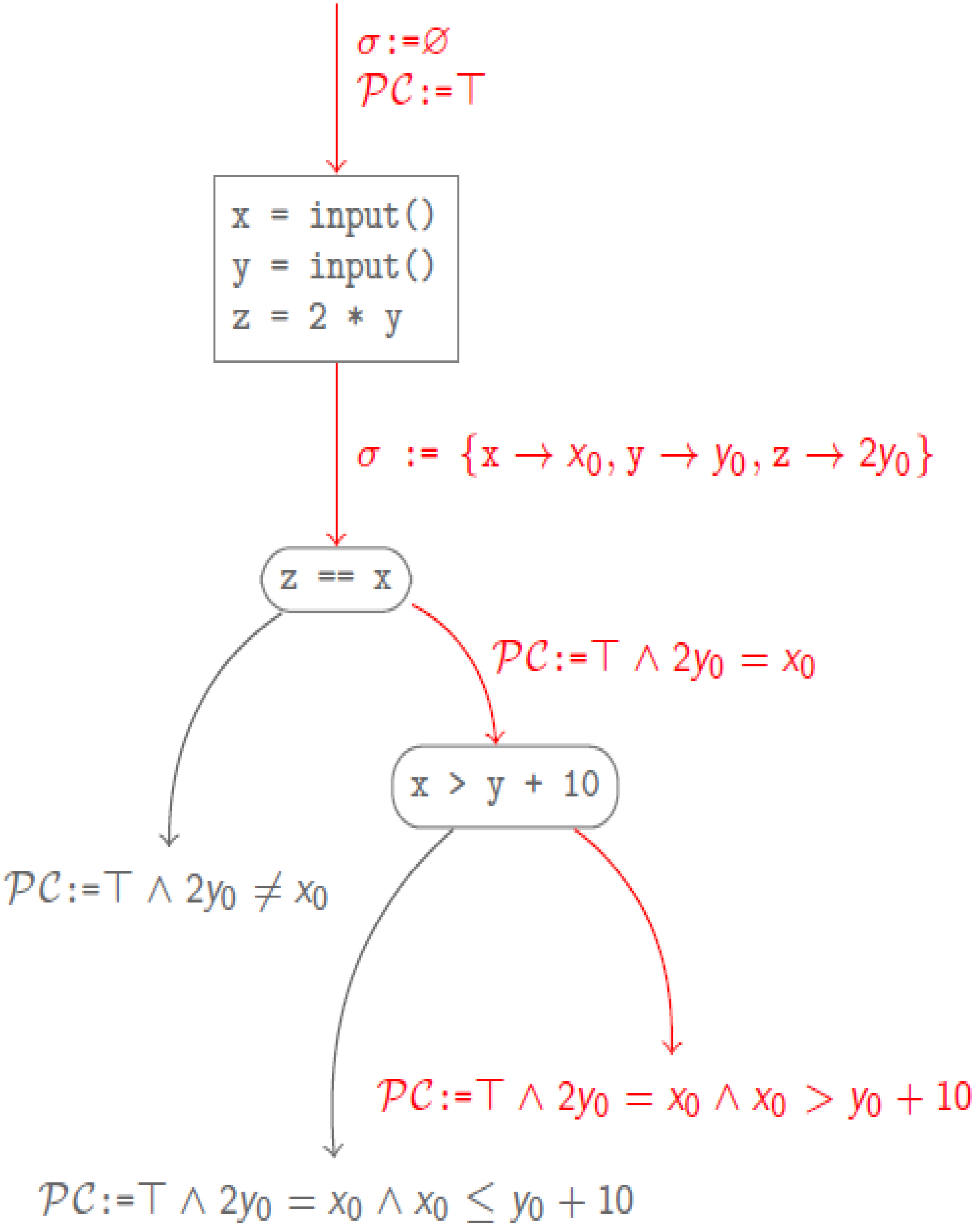
- /\-Solver(T) and /\-Solver(T’) => /\-Solver(TxT’)
- Based on equality propagation

Key 2: DPLL(T)

- /\-Solver(T) => Solver(T)
- Interplay SAT-DPLL and T
- Propag & learning (atomic level)

SYMBOLIC EXECUTION (2005)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```



Given a path of a program

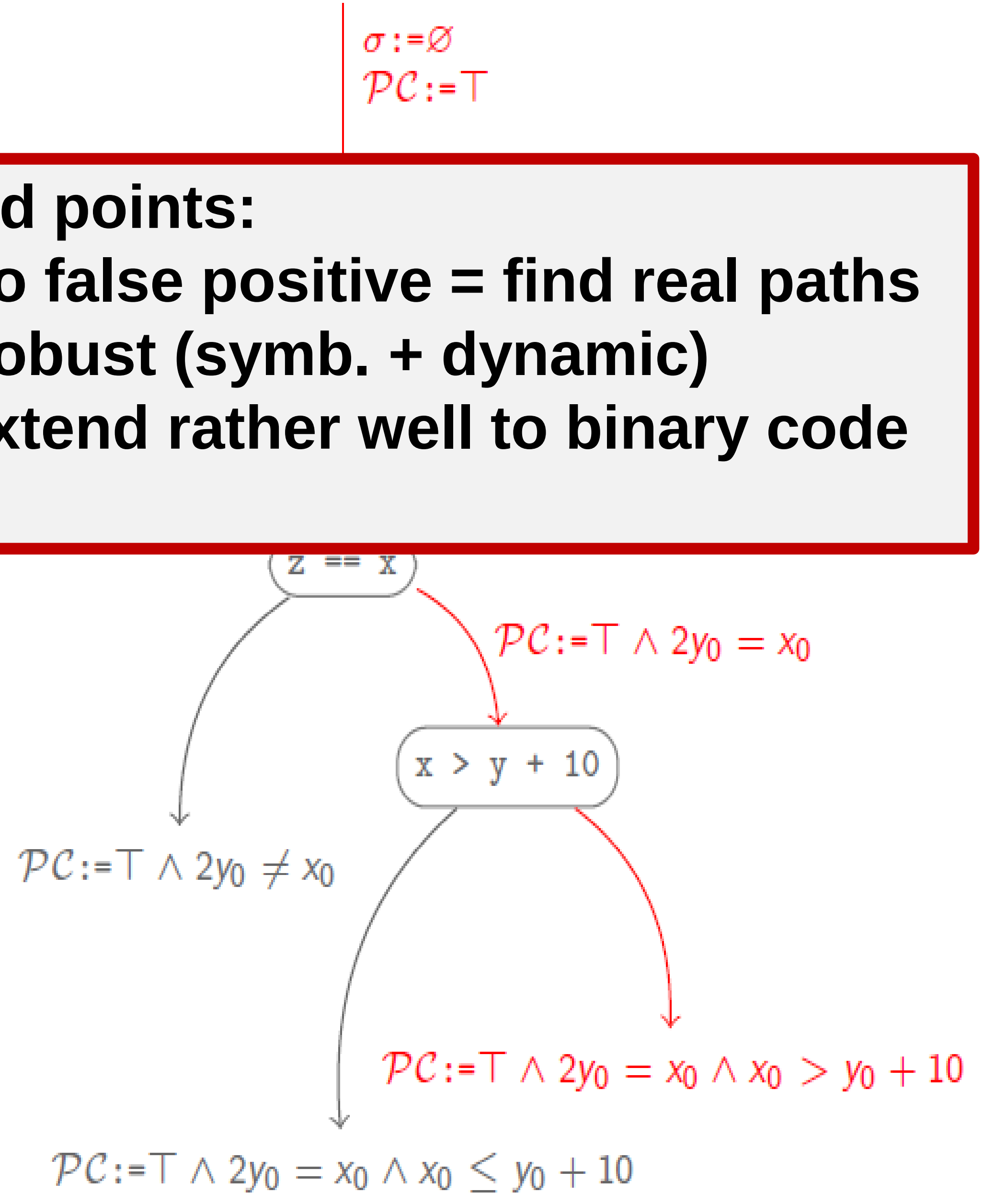
- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

SYMBOLIC EXECUTION (2005)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

Good points:

- **No false positive = find real paths**
- **Robust (symb. + dynamic)**
- **Extend rather well to binary code**



Given a path of a program

- Compute its « path predicate » f
- Solution of f ⬄ input following the path
- Solve it with powerful existing solvers

CURRENT STATE

New challenges

- **New queries: *optimization*, solution counting, supersets of solutions, *soft constraints*, etc.**
- **New logics: fixpoint, come back to full quantifiers**
- **Theories*: float and other large* *finite domains*, memory, etc.**
- **Limits of DPLL(T): *lack of propagation*, shallow combination**

AN OPPORTUNITY FOR CP!!

- **Constraint programming**

- **Good for: finite domains, propagation, optimization, soft constraints**

- **CP & verif: some pionniers**
  - Including A. Gotlieb, B. Marre, M. Ruher
  - It works!
  - Some events: CSTVA, CP meets Verification, Dagstuhl seminar proposal

- **In the following: a few results**
  - Bitvector theory [TACAS 2010, CPAIOR 2017]
  - Float [Marre et al., winner of SMTComp 2017]
  - Arrays [CPAIOR 2012, IJCAI 2017]

BITVECTORS: the problem

Theory of bit-vectors (BV)

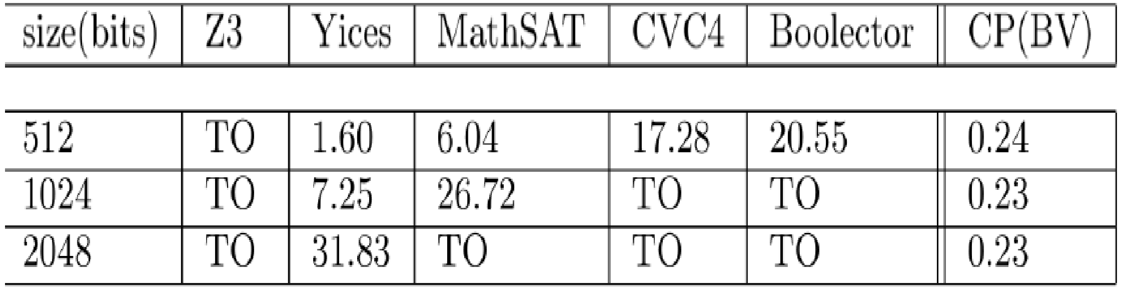
- variables interpreted over fixed-size arrays of bits
- standard low-level operators

- very precise modelling of low-level constructs
- allows multiplication between variables

SMT approach: bitblasting

- (preprocess then) SAT-reduction
- Pros: SAT solvers!
- Cons: miss high-level structure

$$x \times y = (x \mathbin{\&} y) \times (x \mid y) + (x \mathbin{\&} \overline{y}) \times (\overline{x} \mathbin{\&} y)$$

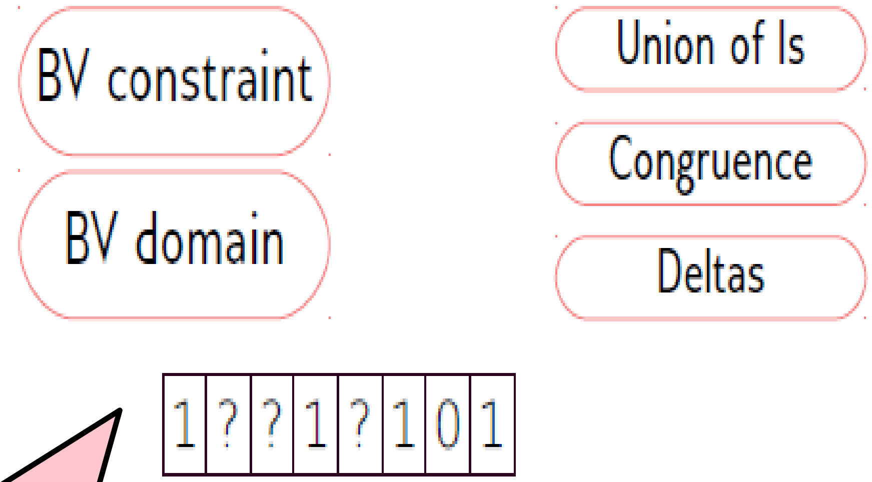


size(bits)	Z3	Yices	MathSAT	CVC4	Boolector	CP(BV)
512	TO	1.60	6.04	17.28	20.55	0.24
1024	TO	7.25	26.72	TO	TO	0.23
2048	TO	31.83	TO	TO	TO	0.23

BITVECTORS & CP: High-level encoding

BVs seen as first-class variables

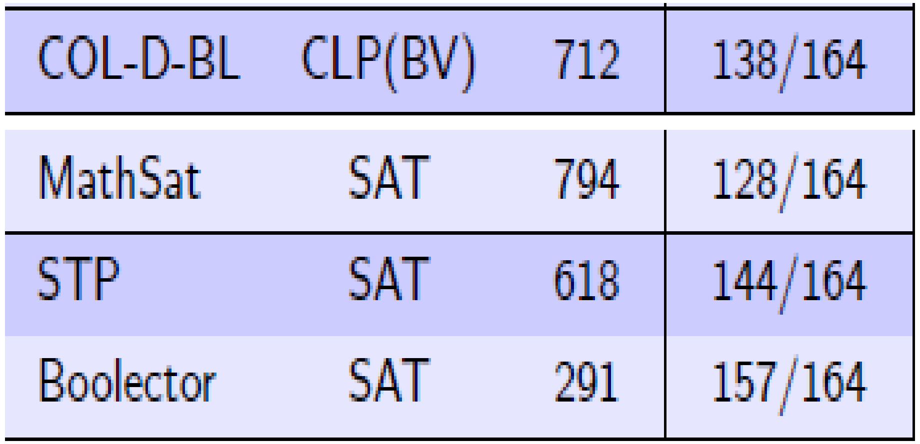
- **Several complementary domains**
- **Local propagation + Inter-domain reduction**
- **Global reasoning**
- **On-the-fly simplification rules**
- **Easy combination with bounded arithmetic**



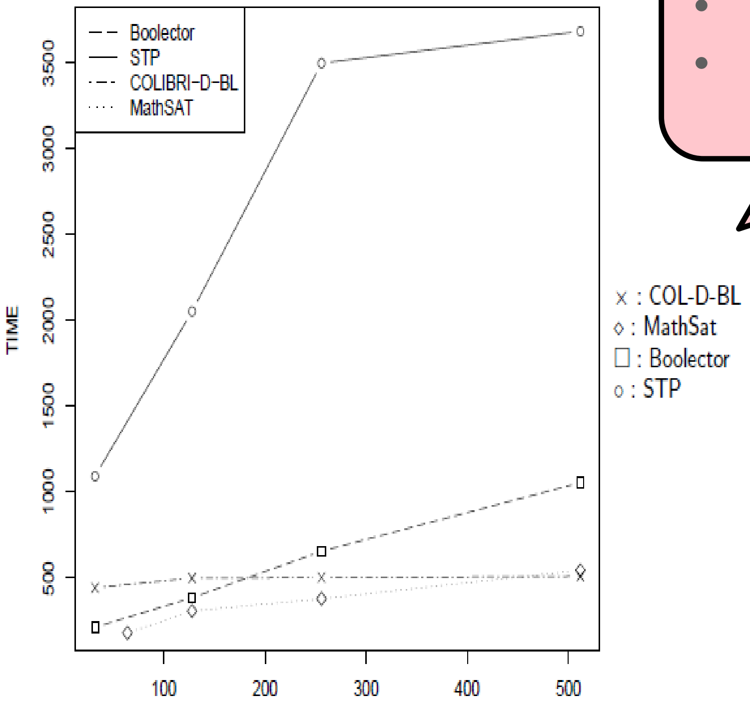
- BV ⇔ intervals
- BV ⇔ congruence
- …

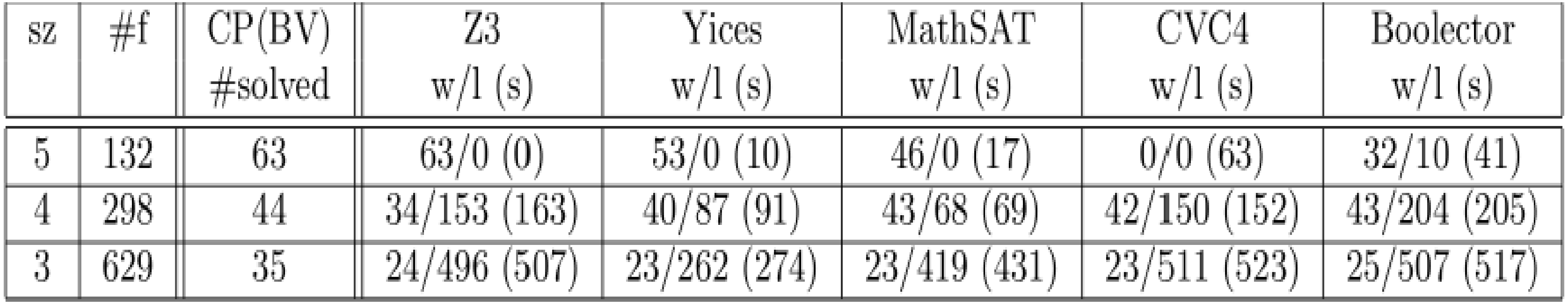
BITVECTORS: results <COLIBRI, B. Marre>

- Fill a large part of the gap
- Complementary



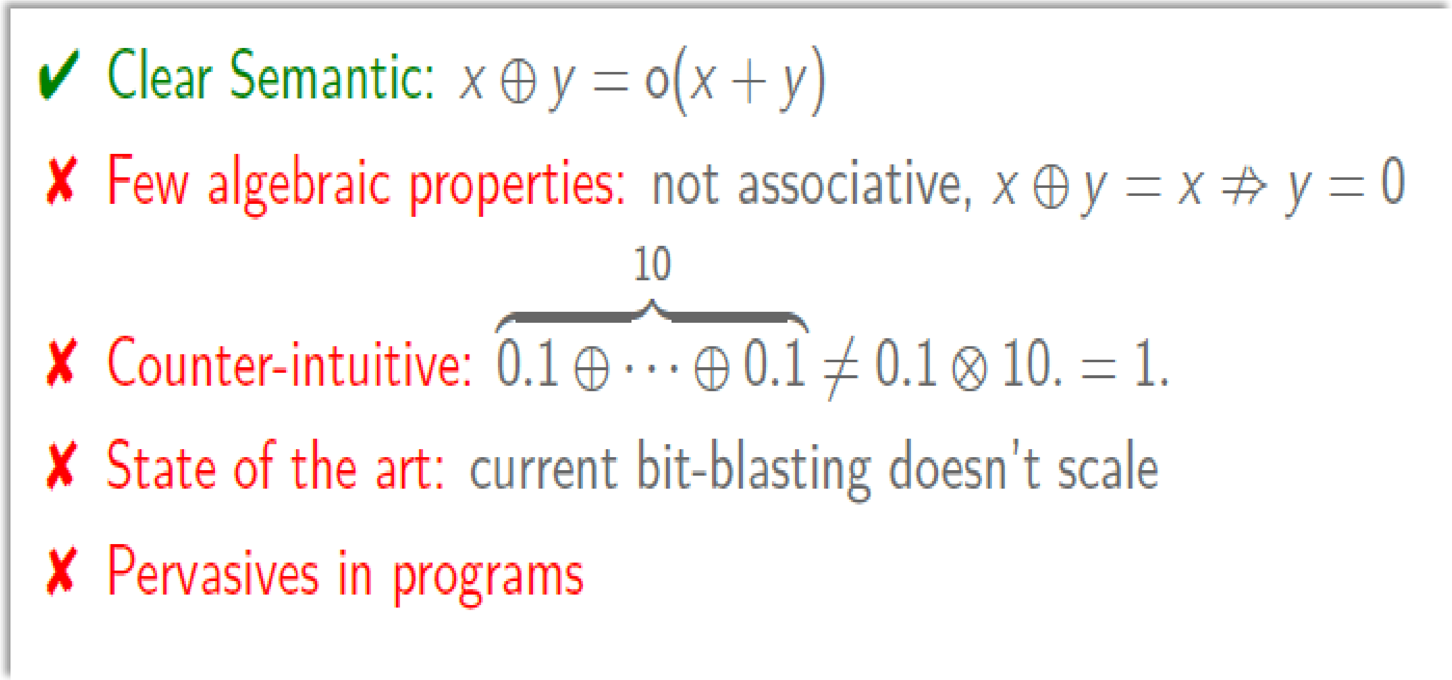
COL-D-BL	CLP(BV)	712	138/164
MathSat	SAT	794	128/164
STP	SAT	618	144/164
Boolector	SAT	291	157/164



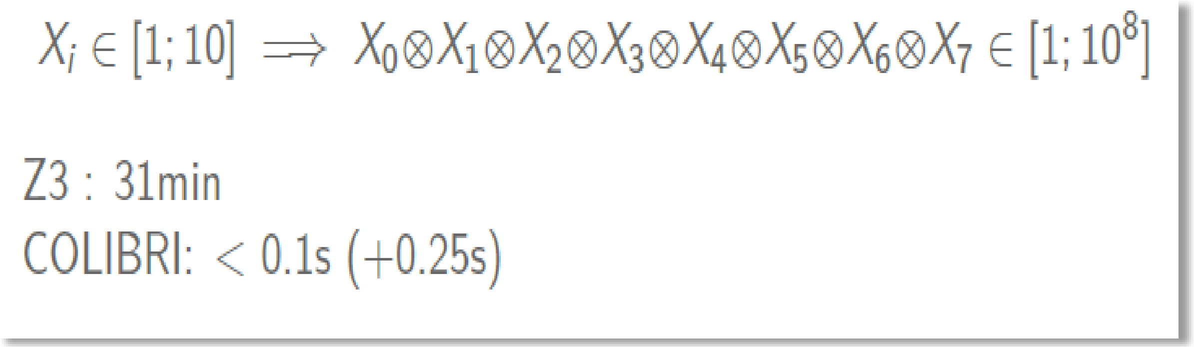


sz	#f	CP(BV) #solved	Z3 w/l (s)	Yices w/l (s)	MathSAT w/l (s)	CVC4 w/l (s)	Boolector w/l (s)
5	132	63	63/0 (0)	53/0 (10)	46/0 (17)	0/0 (63)	32/10 (41)
4	298	44	34/153 (163)	40/87 (91)	43/68 (69)	42/150 (152)	43/204 (205)
3	629	35	24/496 (507)	23/262 (274)	23/419 (431)	23/511 (523)	25/507 (517)

Floats: the problem



- ✓ Clear Semantic: $x \oplus y = o(x + y)$
- ✗ Few algebraic properties: not associative, $x \oplus y = x \not\Rightarrow y = 0$
- ✗ Counter-intuitive: $\overbrace{0.1 \oplus \cdots \oplus 0.1}^{10} \neq 0.1 \otimes 10. = 1.$
- ✗ State of the art: current bit-blasting doesn't scale
- ✗ Pervasives in programs

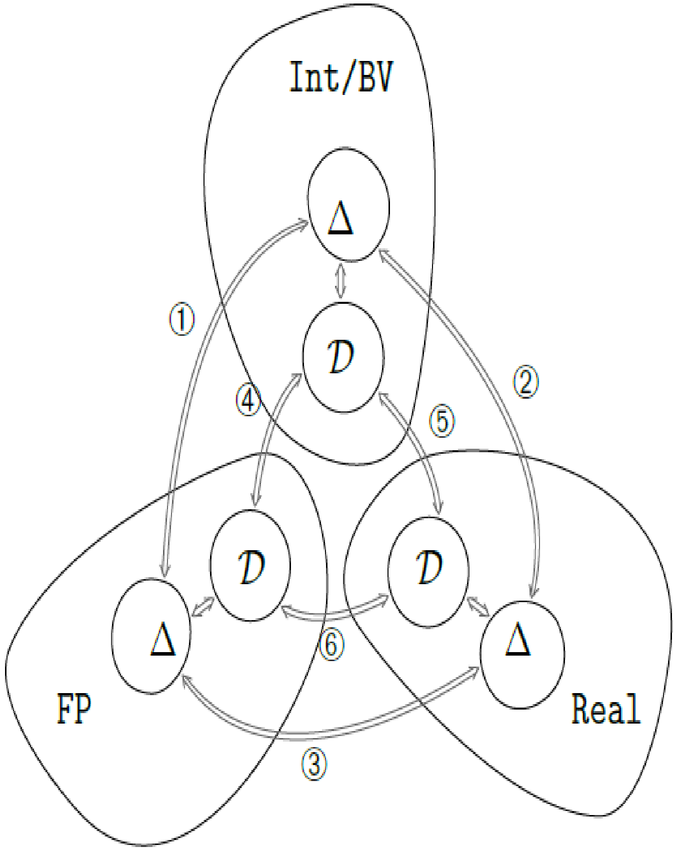


$$X_i \in [1;10] \implies X_0 \otimes X_1 \otimes X_2 \otimes X_3 \otimes X_4 \otimes X_5 \otimes X_6 \otimes X_7 \in [1;10^8]$$

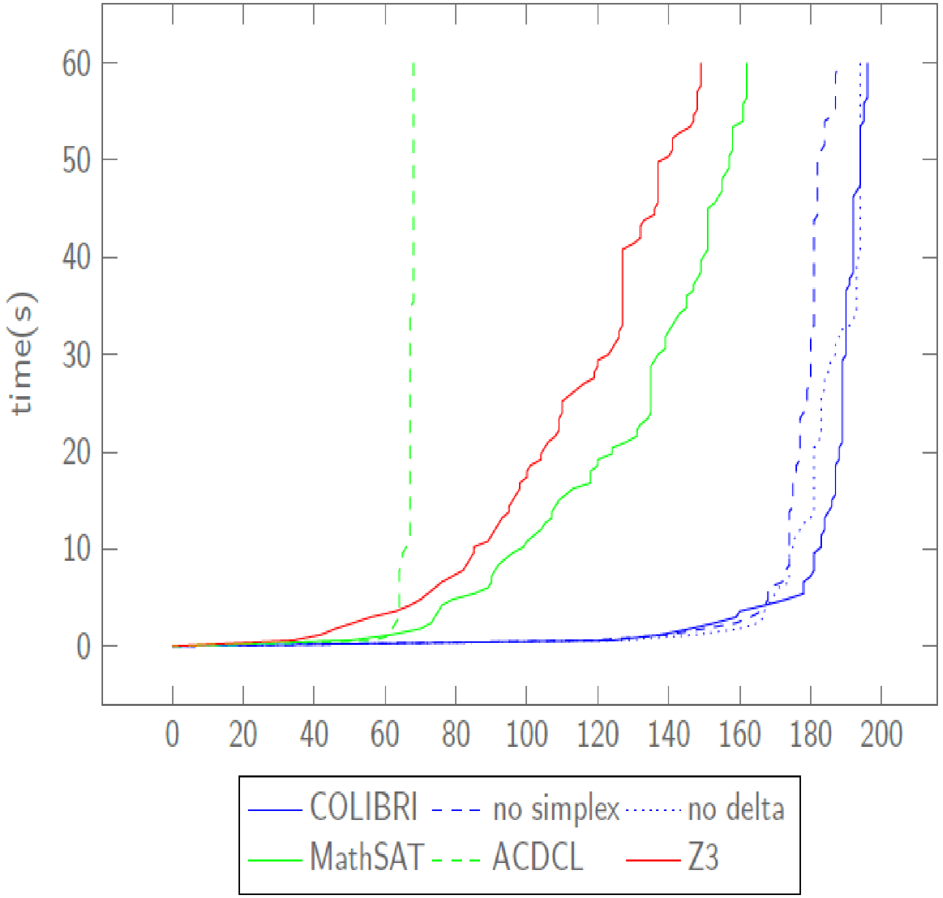
Z3 : 31min
COLIBRI: < 0.1s $(+0.25$s$)$

Floats: High-level encoding, again [Marre et al.]

- Precise domain propagation:
  $x \oplus y = 0.05 \implies x, y \in [-0.1259..; 0.175....]$
  0.05: $0x3fa999999999999a$
- Distance graph on floating-point numbers
- Monotonic functions:
  $o(f(x)) < o(y) \implies o(x) \leq o(f^{-1}(o(y)))$
- Instantiated for many functions
- Linearization of constraints for simplex



Floats: results <COLIBRI>



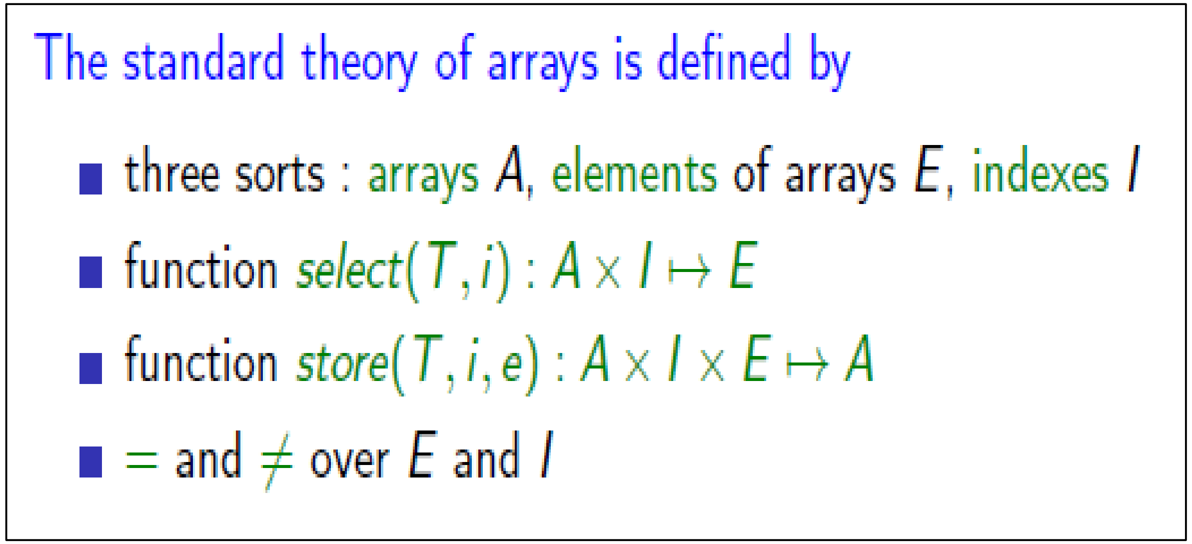
SMTcomp winner

- **FP, FPBV**

Also: AdaCore industrial examples

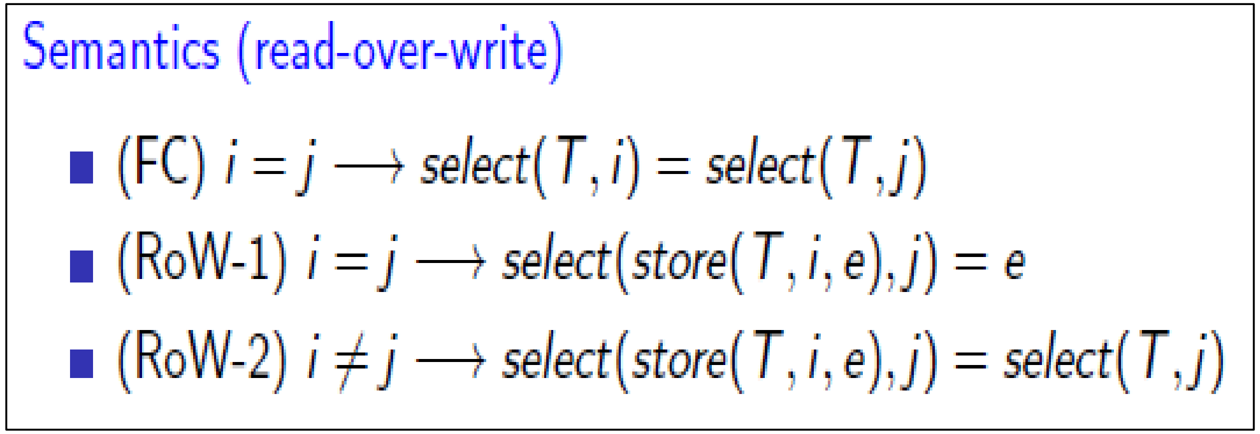
- **FP + BV + integers**

Arrays: the problem



The standard theory of arrays is defined by

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



Semantics (read-over-write)

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

Why?

- **All containers (arrays, vectors, maps)**
- **Memory model**

- NP-complete
- SMT solvers are not good
- No native notion of size

Arrays & CP

Any verification technique needs arrays

- Arbitrary indexes
- Arbitrary size, unbounded, unkown size
- Algos indep. from size
- Combination with bv, floats, integers, etc.

Arrays in CP

- Cons: Fixed size, algo depend on size
- Pros: cheap disjunctive reasoning

Arrays & CP: proposal

Array reduction technique

- Input: extended array formula (size, extensionality, arbitrary combination)
- Output: equisatisfiable fixed-size array formula
- Optimized encoding (break symmetries)
- reuse existing CP algos

Key insights

- Ignore cells, focus on access
- Only = or =!= matters
- Implicit (dis-)equality encoding
- Isolate array reasoning from arith/bv/…

Steps

1. Purification
2. Size elimination
3. Index reduction
4. Size fixing

Arrays & CP: proposal (2)

- Unbounded array

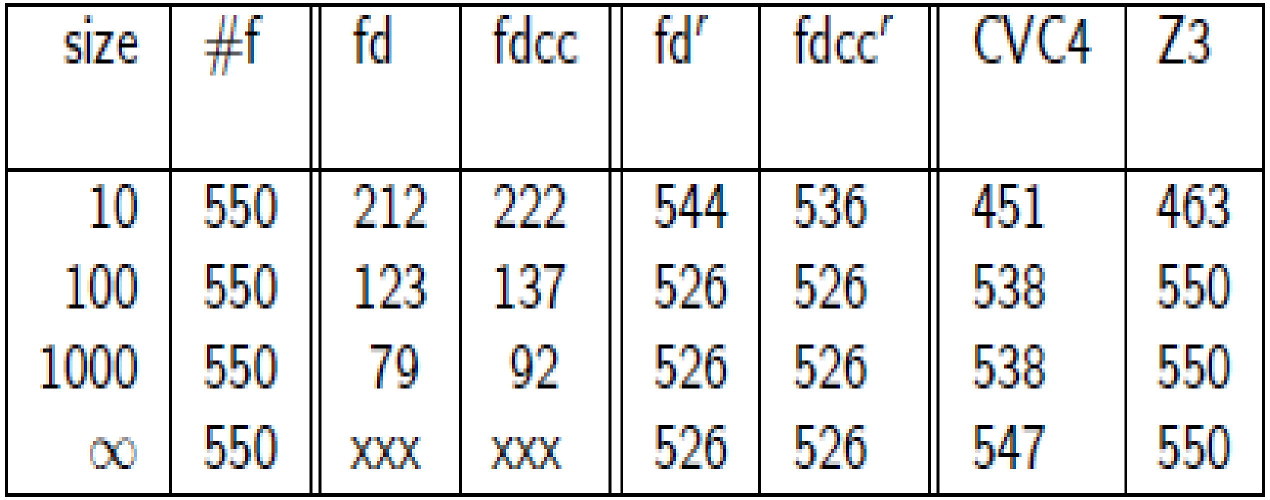
$$\phi \triangleq size(t) = N, t[i] < t[j], i = j$$

- Array of size 2

$$\begin{aligned} \phi^r \triangleq\ & e < f, i = j, i \in 1..s_t, j \in 1..s_t, s_t = N \\ & t^r[i^r] = e,\ t^r[j^r] = f, size(t^r) = 2 \\ & consistent([i,j],[i^r,j^r]), i^r = 1, j^r \in 1..2 \end{aligned}$$

Arrays: results

- CP can handle unbounded arrays
- Close gap with SMT
- Even better for small size



size	#f	fd	fdcc	fd′	fdcc′	CVC4	Z3
10	550	212	222	544	536	451	463
100	550	123	137	526	526	538	550
1000	550	79	92	526	526	538	550
∞	550	xxx	xxx	526	526	547	550

Summary

- **Arrays:**
  - Extend CP techniques to the unbounded case
  - Keep advantage of the propagation
  - Better than SMT on small-size arrays (common in verif)

- **Bitvectors**
  - Propose a novel high-level approach
  - Take advantage of propagation & domain combination
  - Close the gap with SMT, complementary

- **Floats [Marre et al.]**
  - Propose a novel high-level approach
  - Take advantage of propagation & domain combination
  - Won the 2017 SMTCOMP for FPA and FP+BV

OUTLINE

- **Introduction**
- **[past] a tour on software verification & formal methods**
- **[present] logic-based verification, an opportunity for CP**
- **[future] verification & machine learning**
- **Conclusion**

TWO OPPOSITE WAYS OF LIFE …

- **Verification**
  - Need clear specifications
  - Deduction-based
  - Strong guarantees
  - Highly critical systems

- **Machine Learning**
  - Need data
  - Induction-based
  - No guarantees (??)
  - Not-critical systems (??)

Machine Learning for Verification

The sad truth: verification is not that elegant & clean

Many dirty details needs to be set up

- Tools: likely-invariants, hints, parameters (verif vs bug finding – precise vs cost), etc.
- Solvers: which one? Choices of encoding? Parameters (+/- sat or unsat), etc.

Huge difference between expert & naive user

ML could be a key enabler for the adoption of verification

Already a few examples

- Solvers: ML-based portfolio
- SAT Solvers: ML-based branching

Verification for Machine Learning

The frightening truth: ML-enabled critical systems (cars, cobots, laws, etc.)

How to get the promise of ML and still keep strong safety?

- Dynamic programs
- Written in a strange way
- With no spec

Mimic what has been done for standard programs?

Verif could be a key enabler for the adoption of ML in critical systems

Serious challenge!

- Spec?
- operational model + scale
- …

OUTLINE

- Introduction
- [past] a tour on software verification & formal methods
- [present] logic-based verification, an opportunity for CP
- [future] verification & machine learning
- Conclusion

CONCLUSION

- **Software Verification is highly successful**
  - Change the game in highly-regulated fields
  - Start spreading to less critical fields

- **Not so much investigated by the AI community**
  - Yet, constraints & CP can already help verification
  - Fruitful for both Verif & CP

- **Futur: strong relations between Verif & AI?**
  - AI for Verif   ---   Verif for AI
  - In both case: new key enablers and new application domains

Commissariat à l’énergie atomique et aux énergies alternatives
Institut List | CEA SACLAY NANO-INNOV | BAT. 861 – PC142
91191 Gif-sur-Yvette Cedex - FRANCE
www-list.cea.fr

Établissement public à caractère industriel et commercial | RCS Paris B 775 685 019