Embedded software verification by abstract interpretation

Patrick Cousot
CIMS – NYU & ENS

CMACS visit to Rockwell-Collins, Cedar Rapids, Iowa May 3, 2010

Content

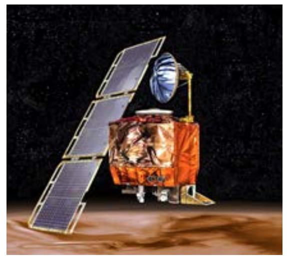
- Motivation for static analysis
- An informal introduction to abstract interpretation
- A short overview of a few applications and on-going work on aerospace software
- References

Motivation

Computer scientists have made great contributions to failures of complex systems

Ariane 5.01 failure (overflow)

Patriot failure (float rounding)



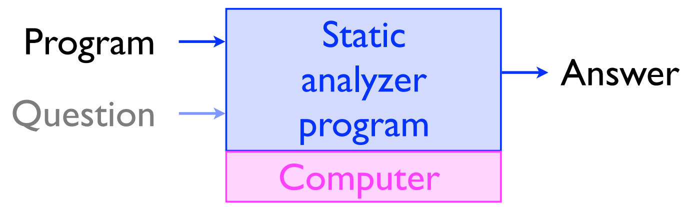
Mars orbiter loss (unit error)

- Onboard checking the presence of bugs is great!
- Proving their absence is even better!!!

Static analysis

Static analysis

- Static analysis consists in automatically answering questions about the runtime executions of programs
- Static means « at compile time », by examining the program text only, without executions on computers
- Automatic means by a computer, without human intervention during the analysis



Static analysis is undecidable

- Undecidability essentially means that any static analyzer/verifier cannot answer "yes" or "no" to a question about all input programs
- It will not terminate or will terminate with answer « I don't know » on infinitely many input programs

Facing undecidability

- Degugging: *test* a few ... many cases → costly, unsafe, not a verification!
- Deductive methods: ask for *human help* (e.g. to make guesses or guide a theorem prover) → complex, error-prone & very costly
- Model checking: explore *finite models* of programs → combinatorial explosion & models may be different from programs
- Abstract interpretation: make *sound approximations* of program executions → always terminate but some potential bug warinings may be false alarms (when the abstraction is incomplete)

Abstract interpretation

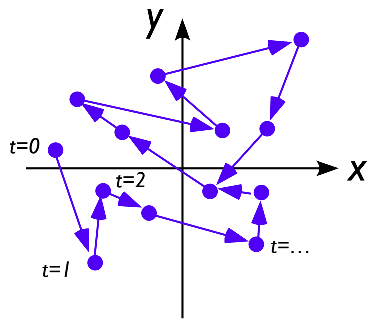
Abstract interpretation

- *Started in the 70's* and well-developped since then
- Originally for inferring program invariants (with first applications to compilation, optimization, program transformation, to help hand-made proofs, etc)
- Based on the idea that undecidability and complexity of automated program analysis can be fought by *approximation*
- Applications evolved from *static analysis* to *verification*
- ***Does scale up!***

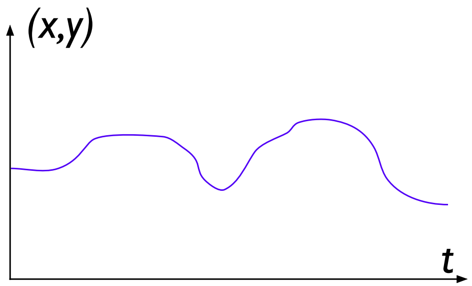
An informal introduction to abstract interpretation

1) Define the programming language semantics

Formalize the concrete ***execution*** *of programs (e.g. transition system)*



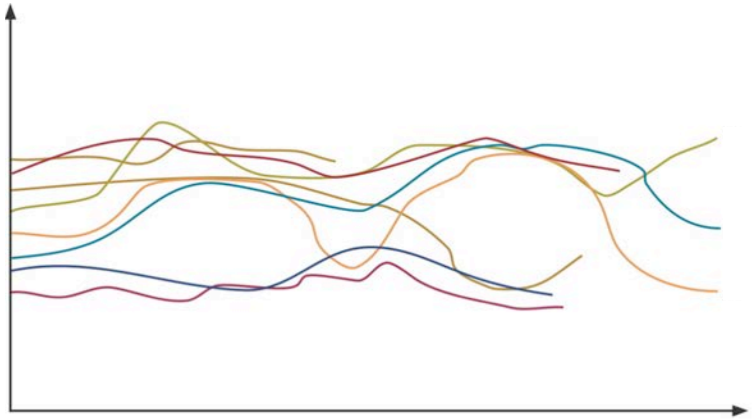
Trajectory
in state space



Space/time trajectory

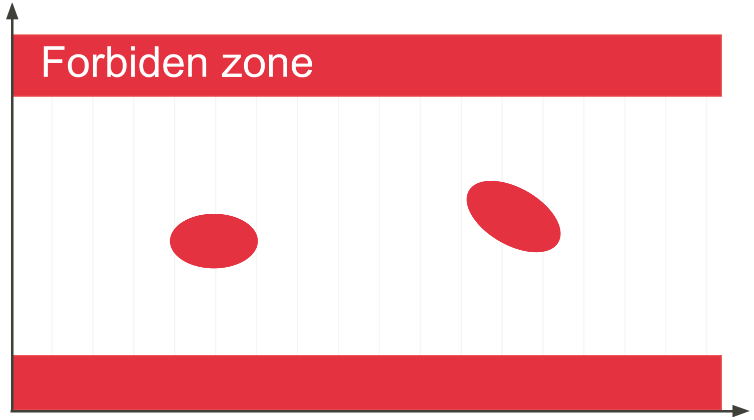
II) Define the program properties of interest

*Formalize what you are interested to **know** about program behaviors*



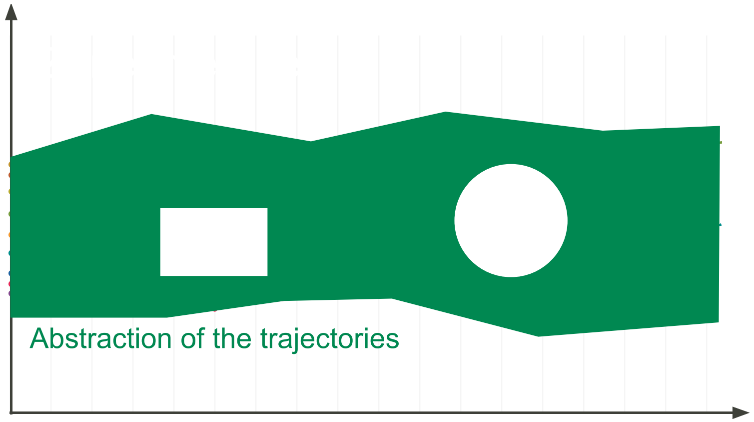
III) Define which specification must be checked

*Formalize what you are interested to **prove** about program behaviors*



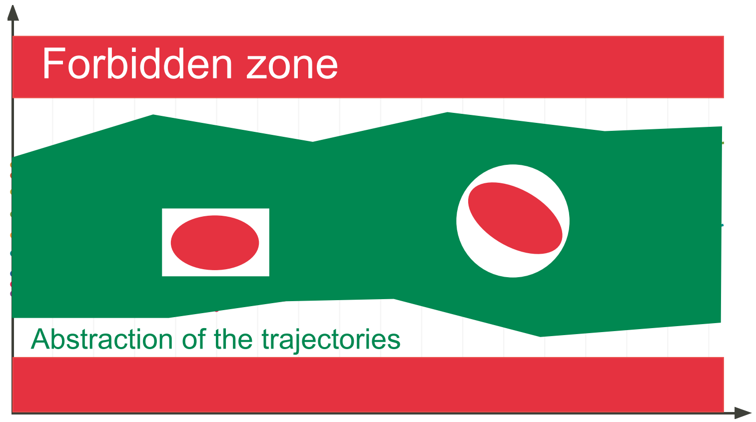
IV) Choose the appropriate abstraction

***Abstract** away all information on program behaviors irrelevant to the proof*



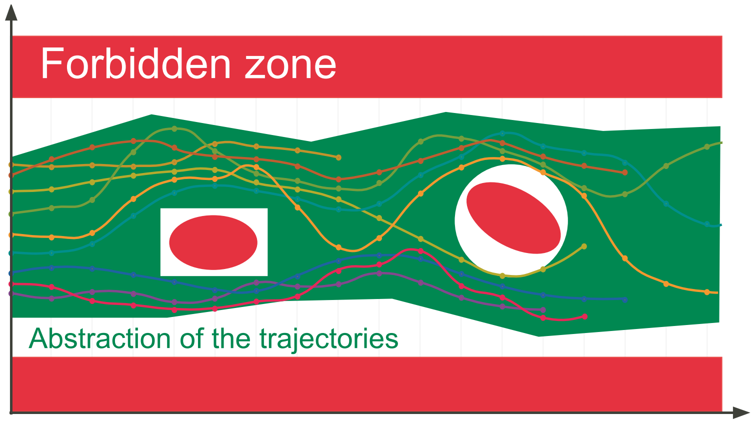
V) Mechanically verify in the abstract

*The proof is fully **automatic***



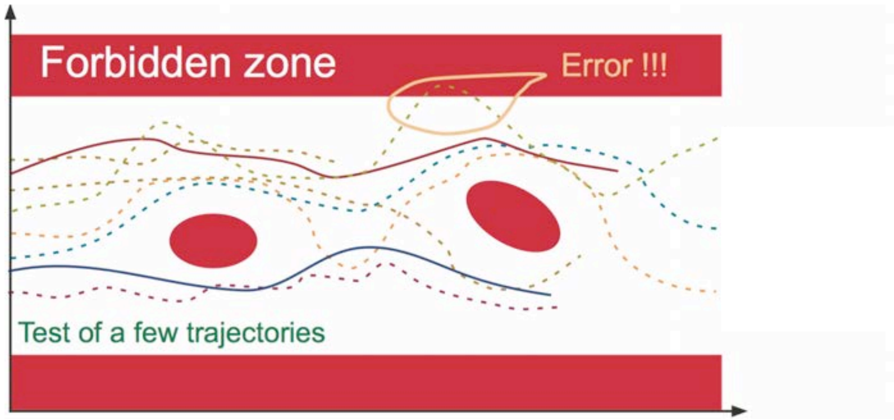
Soundness of the abstract verification

*Never forget any possible case so the **abstract proof is correct in the concrete***



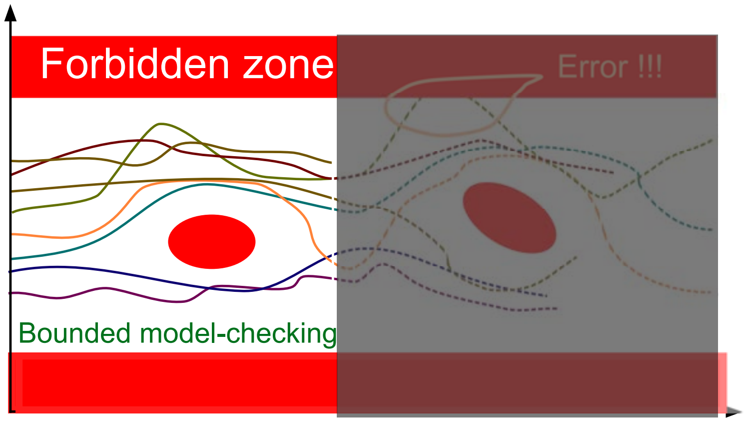
Unsound validation: testing

Try a few cases



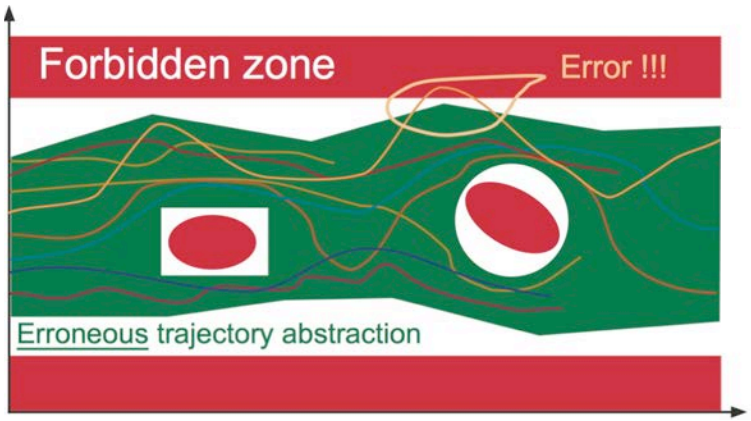
Unsound validation: bounded model-checking

Simulate the beginning of all executions



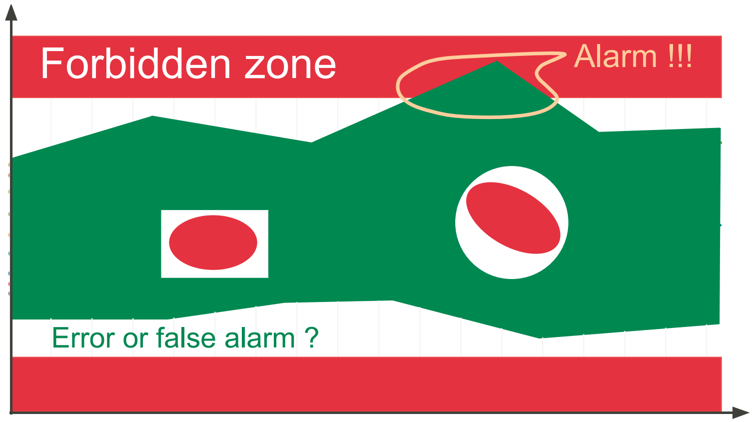
Unsound validation: static analysis

*Many static analysis tools are **unsound** (e.g. Coverity, etc.) so inconclusive*



Incompleteness

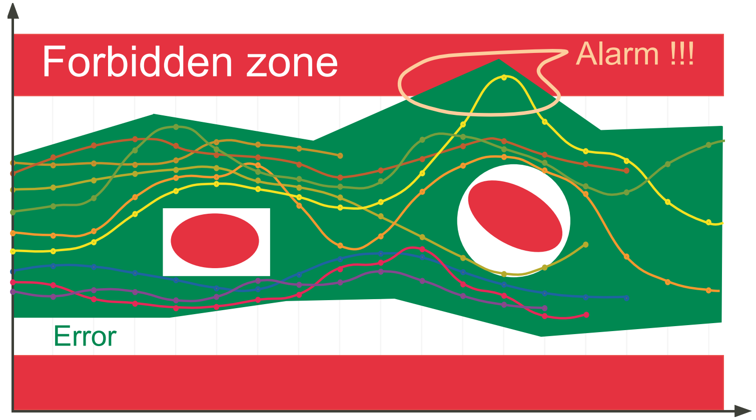
When abstract proofs may fail while concrete proofs would succeed



By soundness an alarm must be raised for this overapproximation!

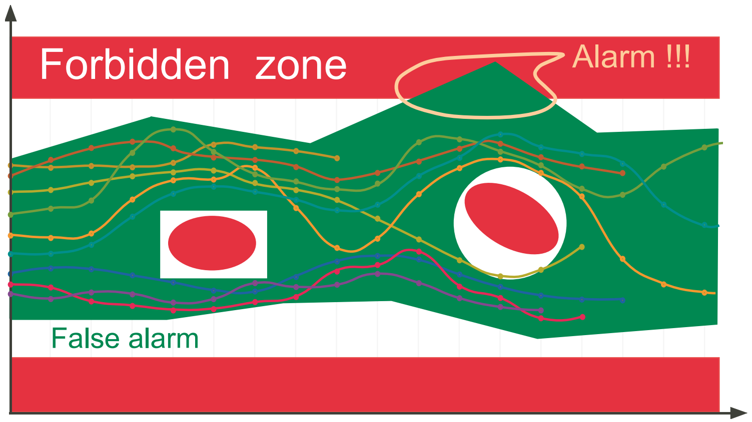
True error

The abstract alarm may correspond to a concrete error



False alarm

The abstract alarm may correspond to no concrete error (false negative)



Principle of an abstract interpreter

- Read the input program
- Optionally read the question (can be implicit e.g. absence of runtime errors or inserted in the program e.g. `assert`)
- Compute the abstraction of the program execution
- Output the result:
  - Answer to the question (yes, no, I don't know)
  - Optionally, provide information on program execution (e.g. over-approximation of the range of variation of numerical variables, shape of data structures, etc)

What to do about false alarms?

- Automatic refinement: inefficient and may not terminate (Gödel)
- Domain-specific abstraction:
  - Adapt the abstraction to the *programming paradigms* typically used in given *domain-specific applications*
  - e.g. *synchronous control/command*: no recursion, no dynamic memory allocation, maximum execution time, etc.

ASTRÉE

Target language and applications

- C programming language
  - Without recursion, `longjump`, dynamic memory allocation, conflicting side effects, backward jumps, system calls (stubs)
  - With all its horrors (`union`, pointer arithmetics, etc)
  - Reasonably extending the standard (e.g. size & endianess of integers, IEEE 754-1985 floats, etc)
- Synchronous control/command
  - e.g. generated from Scade/Lustre, Simulink, or a proprietary system

The class of considered periodic synchronous programs

```
declare volatile input, state and output variables;
initialize state and output variables;
loop forever
   - read volatile input variables,
   - compute output and state variables,
   - write to output variables;
  __ASTREE_wait_for_clock ();
end loop
```

Task scheduling is static:

– Requirements: the only interrupts are clock ticks;

– Execution time of loop body less than a clock tick, as verified by the aiT WCET Analyzers

The semantics of C implementations is very hard to define

What is the effect of out-of-bounds array indexing?

```
% cat unpredictable.c
#include <stdio.h>
int main () { int n, T[1];
 n = 2147483647;
 printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

Yields different results on different machines:

`n = 2147483647, T[n] = 2147483647`	Macintosh PPC
`n = 2147483647, T[n] = -1208492044`	Macintosh Intel
`n = 2147483647, T[n] = -135294988`	PC Intel 32 bits
`Bus error`	PC Intel 64 bits

Implicit specification

- Absence of runtime errors: overflows, division by zero, buffer overflow, null & dangling pointers, alignment errors, …
- Semantics of runtime errors:
  1. Terminating execution: stop (e.g. floating-point exceptions when traps are activated)
  2. Predictable outcome: go on with worst case (e.g. signed integer overflows result in some integer, some options: e.g. modulo arithmetics)
  3. Unpredictable outcome: stop on error (e.g. memory corruption), go on with non-erroneous cases

Example of error with predictable output: modular arithmetics

```
% cat -n modulo-c.c
     1 #include <stdio.h>
     2 int main () {
     3 int x,y;
     4 x = -2147483647 / -1;
     5 y = ((-x) -1) / -1;
     6 printf("x = %i, y = %i\n",x,y);
     7 }
     8

% gcc modulo-c.c
% ./a.out
x = 2147483647, y = -2147483648
```

positive

negative

Analysis by ASTRÉE

```
% cat -n modulo.c
     1 int main () {
     2 int x,y;
     3 x = -2147483647 / -1;
     4 y = ((-x) -1) / -1;
     5 __ASTREE_log_vars((x,y));
     6 }
     7
% astree -exec-fn main -unroll 0 modulo.c\
 |& egrep -A 1 "(<integers)|(WARN)"
modulo.c:4.4-18::[call#main@1:]: WARN: signed int arithmetic range
  {2147483648} not included in [-2147483648, 2147483647]
  <integers (intv+cong+bitfield+set): y in [-2147483648, 2147483647] /\ Top
   x in {2147483647} /\ {2147483647} >
```

ASTRÉE signals the overflow and goes on with an unknown integer (as required by the C standard)

Example of error with predictable output: float arithmetics

```
% cat -n overflow.c
 1  void main () {
 2  double x,y;
 3  x = 1.0e+256 * 1.0e+256;
 4  y = 1.0e+256 * -1.0e+256;
 5  __ASTREE_log_vars((x,y));
 6  }
% gcc overflow.c
% ./a.out
x = inf, y = -inf
```

```
% astree -exec-fn main
overflow.c |& grep "WARN"
overflow.c:3.4-23::[call#main1:]:
WARN: double arithmetic range
[1.79769e+308, inf] not
included in [-1.79769e+308,
1.79769e+308]
overflow.c:4.4-24::[call#main1:]:
WARN: double arithmetic range
[-inf, -1.79769e+308] not
included in [-1.79769e+308,
1.79769e+308]
```


Example of error with <u>un</u>predictable output: buffer overflow

```
% cat -n unpreditable-a.c
     1  const int false = 0;
     2  int main () { int n, T[1], x;
     3  n = 1;
     4  x = T[n];
     5  __ASTREE_assert((false));
     6  }
% astree -exec-fn main unpreditable-a.c |& grep "WARN"
unpreditable-a.c:4.4-8::[call#main@2:]: WARN: invalid dereference: dereferencing
4 byte(s) at offset(s) [4;4] may overflow the variable T of byte-size 4
%
```

No alarm on `assert(false)` because execution is assumed to stop after a definite runtime error with unpredictable results [(4)].

(4) Equivalent semantics if no alarm.

Soundness

- In absence of error of type 3. (without unpredictable consequences) $\longrightarrow$ fully sound
- In presence of errors of type 3. (with unpredictable consequences), ASTRÉE may miss further errors occuring *after this first error due to the unpredictable behavior* $\longrightarrow$ sound up to the first error with unpredictable consequences

Rounding is not an error but is problematic!

```
/* float-error.c */
int main () {
  float x, y, z, r;
  x = 1.000000019e+38;
  y = x + 1.0e21;
  z = x - 1.0e21;
  r = y - z;
  printf("%f\n", r);
}
% gcc float-error.c
% ./a.out
0.000000
```

```
/* double-error.c */
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951488.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
134217728.000000
```

$$(x + a) - (x - a) \neq 2a$$

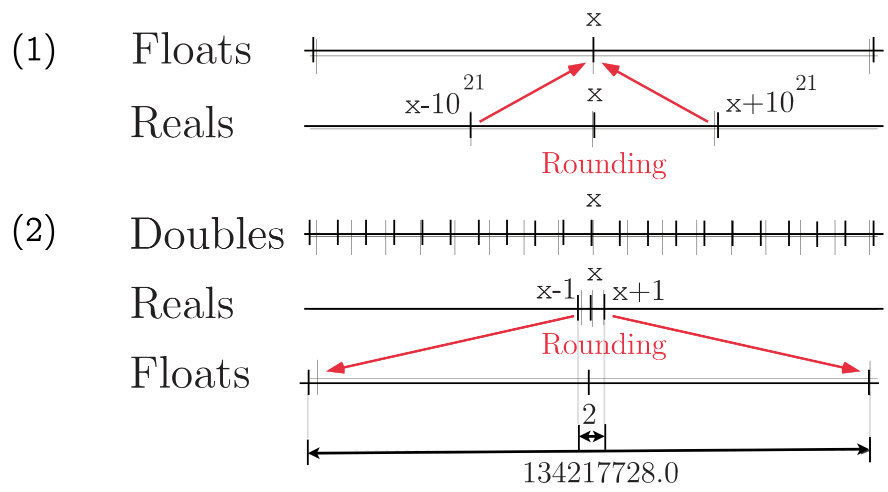
Rounding is not an error but is problematic!

```
/* float-error.c */
int main () {
  float x, y, z, r;
  x = 1.000000019e+38;
  y = x + 1.0e21;
  z = x - 1.0e21;
  r = y - z;
  printf("%f\n", r);
}
% gcc float-error.c
% ./a.out
0.000000
```

```
/* double-error.c */
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951487.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
0.000000
```

$$(x + a) - (x - a) \neq 2a$$

Explanation of the huge rounding *error*

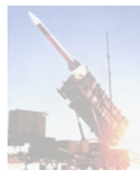


Analysis by ASTRÉE

```
% cat -n double-error.c
 2  int main () {
 3  double x; float y, z, r;;
 4  /* x = ldexp(1.,50)+ldexp(1.,26); */
 5  x = 1125899973951488.0;
 6  y = x + 1;
 7  z = x - 1;
 8  r = y - z;
 9  __ASTREE_log_vars((r));
10  }
% gcc double-error.c
% ./a.out
134217728.000000
% astree -exec-fn main -print-float-digits 10 double-error.c |& grep "r in
direct = <float-interval: r in [-134217728, 134217728] >
```

[13] ASTRÉE makes a worst-case assumption on the rounding ($+\infty$, $-\infty$, 0, nearest) hence the possibility to get -134217728.

Example of accumulation of rounding errors



```
% cat -n rounding-c.c
 1  #include <stdio.h>
 2  int main () {
 3   int i; double x; x = 0.0;
 4   for (i=1; i<=1000000000; i++) {
 5    x = x + 1.0/10.0;
 6   }
 7  printf("x = %f\n", x);
 8  }
% gcc rounding-c.c
% ./a.out
x = 99999998.745418
%
```

since $(0.1)_{10} = (0.0001100110011001100\ldots)_2$

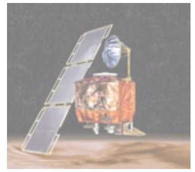
Analysis by ASTRÉE

```
% cat -n rounding.c
     1  int main () {
     2   double x; x = 0.0;
     3   while (1) {
     4    x = x + 1.0/10.0;
     5    __ASTREE_log_vars((x));
     6    __ASTREE_wait_for_clock(());
     7   }
     8  }
% cat rounding.config
 __ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem rounding.config -unroll 0 rounding.c\
  |& egrep "(x in)|(\|x\|)|(WARN)" | tail -2
direct = <float-interval: x in [0.1, 200000040.938] >
  |x| <= 1.*((0. + 0.1/(1.-1))*(1.)^clock - 0.1/(1.-1)) + 0.1
      <= 200000040.938
```


Scaling is not an error but can be problematic

```
% cat -n scale.c
 1 int main () {
 2  float x; x = 0.70000001;
 3  while (1) {
 4   x = x / 3.0;
 5   x = x * 3.0;
 6   __ASTREE_log_vars((x));
 7   __ASTREE_wait_for_clock(());
 8  }
 9 }
```

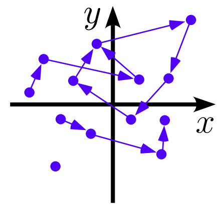
```
% gcc scale.c
% ./a.out
x = 0.699999988079071
```



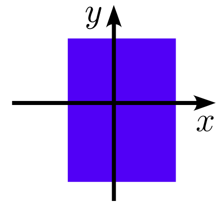
```
% cat scale.config
 __ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem scale.config -unroll 0 scale.c\
 |& grep "x in" | tail -1
direct = <float-interval: x in [0.69999986887, 0.700000047684] >
%
```


Examples of abstractions in ASTRÉE

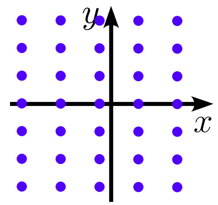
Abstractions



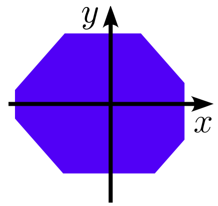
Collecting semantics: partial traces



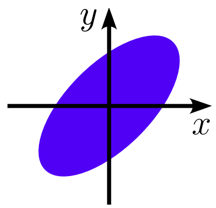
Intervals: $\mathtt{x} \in [a, b]$



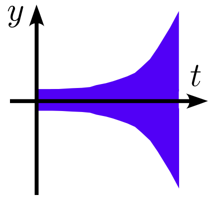
Simple congruences: $\mathtt{x} \equiv a[b]$



Octagons: $\pm\mathtt{x} \pm \mathtt{y} \leqslant a$



Ellipses: $\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{x}\mathtt{y} \leqslant d$



Exponentials: $-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

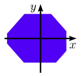
Example of general purpose abstraction: octagons

- Invariants of the form $\pm x \pm y \leq c$, with $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N^3)$ time cost.
- Example:

```
while (1) {
  R = A-Z;
  L = A;
  if (R>V)
    { ★ L = Z+V; }
  ★
}
```

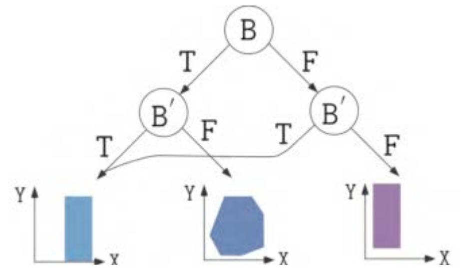
- At ★, the interval domain gives L ≤ max(max A, (max Z)+(max V)).
- In fact, we have L ≤ A.
- To discover this, we must know at ★ that R = A-Z and R > V.

- Here, R = A-Z cannot be discovered, but we get L-Z ≤ max R which is sufficient.
- We use many octagons on **small packs** of variables instead of a large one using all variables to cut costs.



Example of general purpose abstraction: decision trees

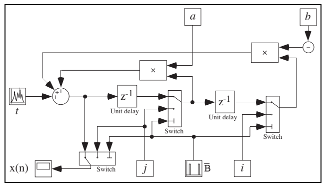
```
/* boolean.c */
typedef enum {F=0,T=1} BOOL;
BOOL B;
void main () {
  unsigned int X, Y;
  while (1) {
    ...
    B = (X == 0);
    ...
    if (!B) {
      Y = 1 / X;
    }
    ...
  }
}
```



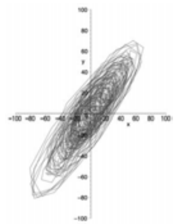
The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leaves

Filters

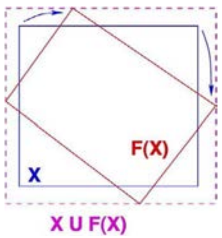
2d Order Digital Filter:



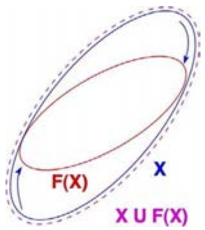
- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.



execution trace



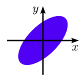
unstable interval



stable ellipsoid

Example of domain-specific abstraction: ellipses

```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
            + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```



Example of domain-specific abstraction: exponentials

```
% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I; int R; BOOLEAN T;
void main() {
  R = 0;
  while (TRUE) {
    __ASTREE_log_vars((R));
    if (I) { R = R + 1; }          ← potential overflow!
    else { R = 0; }
    T = (R >= 100);
    __ASTREE_wait_for_clock(());
  }}
% cat count.config
__ASTREE_volatile_input((I [0,1]));
__ASTREE_max_clock((3600000));
% astree -exec-fn main -config-sem count.config count.c|grep '|R|'
|R| <= 0. + clock *1. <= 3600001.
```


Example of domain-specific abstraction: exponentials

```
% cat retro.c
typedef enum {FALSE=0, TRUE=1} BOOL;
BOOL FIRST;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;

void dev( )
{ X=E;
  if (FIRST) { P = X; }
  else
    { P =  (P - ((((2.0 * P) - A) - B)
            * 4.491048e-03)); };
  B = A;
  if (SWITCH) {A = P;}
  else {A = X;}
}
```

```
void main()
{ FIRST = TRUE;
  while (TRUE) {
    dev( );
    FIRST = FALSE;
    __ASTREE_wait_for_clock(());
  }}
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));
|P| <= (15.  + 5.87747175411e-39
/ 1.19209290217e-07) * (1 +
1.19209290217e-07)^clock - 5.87747175411e-39
/ 1.19209290217e-07 <= 23.0393526881
```


Arithmetic-geometric abstraction

– Abstract domain: $(\mathbb{R}^+)^5$

– Concretization:

$\gamma \in (\mathbb{R}^+)^5 \longmapsto \wp(\mathbb{N} \mapsto \mathbb{R})$

$\gamma(M, a, b, a', b') =$

$$\{f \mid \forall k \in \mathbb{N} : |f(k)| \leq \left(\boldsymbol{\lambda}\, x \cdot ax + b \circ (\boldsymbol{\lambda}\, x \cdot a'x + b')^k\right)(M)\}$$

i.e. any function bounded by the arithmetic-geometric progression.

An erroneous common belief on static analyzers

"The properties that can be proved by static analyzers are often simple" [2]

Like in mathematics:

- May be simple to state (no overflow)
- But harder to discover (`S[0], S[1] in [-1327.02698354, 1327.02698354]`)
- And difficult to prove (since it requires finding a non trivial non-linear invariant for second order filters with complex roots [Fer04], which can hardly be found by exhaustive enumeration)

Reference

[2] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 7, July 2008.

Industrial applications

Examples of applications

- Verification of the absence of runtime-errors in
  - Fly-by-wire flight control systems

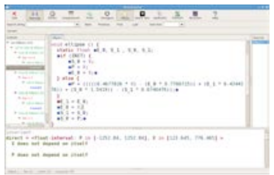
  - ATV docking system

  - Flight warning system
    (on-going work)

Industrialization

- 8 years of research (CNRS/ENS/INRIA):

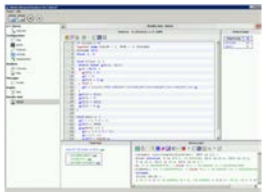
  www.astree.ens.fr

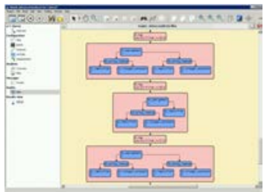


- Industrialization by AbsInt (since Jan. 2010):

  www.absint.com/astree/

  AbsInt Angewandte Informatik





Example of case study: the ATV docking control software

O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, M. Turin.
Space software validation using Abstract Interpretation.
Proc. 13[th] Data Systems in Aerospace, DASIA 2009, Istanbul, Turkey, 26-29 May 2009, © Eurospace, Paris.

The ASTRIUM ST case study MSU SW

- The MSU SW contains mainly:
  1. Navigation and control algorithms
  2. A (very simplified) mission management
- Single task cyclic synchronous software
- Initially in ADA → Scade 5/6 exact model → C (38K LOCS)

“ ⟹ **The C code of the case study may contain errors!** ” [:-)]

Preparatory work

- Definition of stubs for the library (reusable)
- Choice of code generation options for SCADE
- Definition of a few environment properties (a few input ranges)
- Setting ASTRÉE parameters
- Analysis takes < 4mn

Alarms raised by ASTRÉE

- Complex control flow
- Quaternion computation (normalisation)
- Runge Kutta integration (4th order integration scheme)
- Kalman filters (8th order linear filter)
- Controller estimation

} False alarms

- Bugs

} True alarms

Analysis of the alarms

- Correct bugs (in Scade compiler V6 and in the analyzed program)
- Add numerical protections on environment (forgotten hypotheses)
- Add numerical protections in computations (in a first phase)

➔ **0 false alarms**

Very efficient tool compared to Polyspace Verifier

Resolution of the alarms

- Manual first phase:
  - Complex control flow } Improvement of the model
  - Quaternion computation
  - Runge Kutta integration
  - Kalman filters
  - Controller estimation

  (Quaternion computation … Controller estimation) } ≈ 30 numerical protections
  - Bugs } Corrected

➔ 0 false alarm

- Automatic second phase:

Design and implementation of a few abstract domains (quaternions, Runge Kutta integration, Kalman filters) avoiding the unecessary protections **➔ 0 false alarm**

Example of new abstract domain: quaternions

- Quaternions $q = (u, i, j, k)$ are a number system extending complex numbers and applied to mechanics in three-dimensional space
- ASTRÉE did not handle precisely enough the normalization $||q|| \triangleq \sqrt{u^2 + i^2 + j^2 + k^2}$
- An new abstract domain added to ASTRÉE solved this problem
- The abstract properties are $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$

where are 4 variables and is an interval meaning $\sqrt{\mathtt{x}_1^2 + \mathtt{x}_2^2 + \mathtt{x}_3^2 + \mathtt{x}_4^2} \in I$

On-going work

Verification of target programs

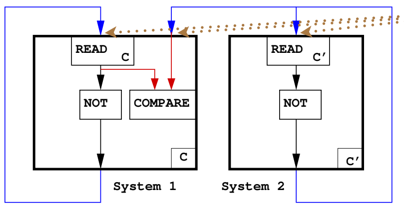
Verification of compiled programs

- The valid source may be proved correct while the certified compiler is incorrect so the target program may go wrong
- Possible approaches:
  - Verification at the target level
  - Source to target proof translation and proof check on the target
  - ✳ Translation validation (local verification of equivalence of run-time error free source and target)
  - Formally certified compilers

Verification of imperfectly clocked synchronous systems

Imperfect synchrony

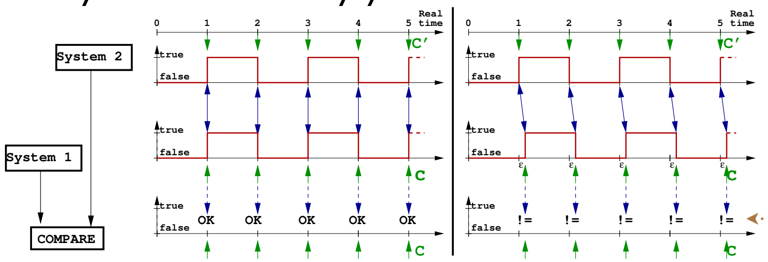
- Example of (buggy) communicating synchronous systems:



blackboard inputs

- negate previous input (on clocks C and C')
- compare inputs

- Synchronized and dysynchronized executions:



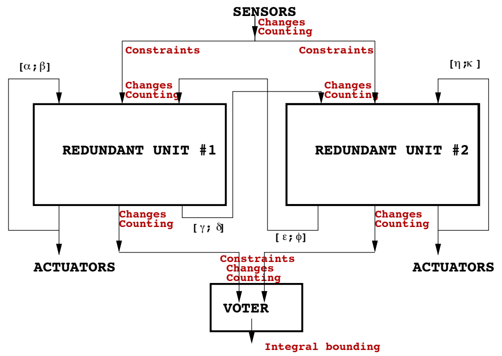
flawed alarms

Semantics and abstractions

- Continuous semantics (value *s(t)* of signals *s* at any time *t*)
- Clock ticks and serial communications do happen in known time intervals $[l, h], l \leq h$
- Examples of abstractions:
  - $\forall t \in [a; b] : s(t) = x.$
  - $\exists t \in [a; b] : s(t) = x.$
  - change counting $(\leqslant k, a \blacktriangleright \blacktriangleleft b)$ and $(\geqslant k, a \blacktriangleright \blacktriangleleft b)$

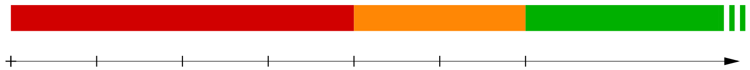
    (signal changes less (more) than *k* times in time interval $[a, b]$)

Example of static analysis



For how long should the input be stabilized before deciding on disagreement?

Specification : no alarm raised with a normal input



0 | 2/3 Δ | Δ

input stability $< \Delta$: counter-example | Between $\frac{2}{3} \times \Delta$ and Δ : ? | input stability $> \Delta$: the analyzer proves the specification

Formal verification of static analyzers

- Intensive work on formalizing the theory of abstract interpretation in Coq
- Proofs essentially done by hand
- Presently verify the correctness of the implementation of abstract domains (e.g. intervals, octagons, ...)
- Then consider combinations of abstract domains
- Ultimately *might* be able to consider the whole static analyzer

THÉSÉE: Verification of embedded real-time parallel C programs

Parallel programs

- Bounded number of processes with shared memory, events, semaphores, message queues, blackboards,…
- Processes created at initialization only
- Real time operating system (ARINC 653) with fixed priorities (highest priority runs first)
- Scheduled on a single processor

Verified properties

- Absence of runtime errors
- Absence of unprotected data races

Semantics

- No memory consistency model for C
- Optimizing compilers consider sequential processes out of their execution context

init: flag1 = flag2 = 0

process 1:	process 2:
`flag1 = 1;`	`flag2 = 1;`
`if (!flag2)`	`if (!flag1)`
`{`	`{`
`/* critical section */`	`/* critical section */`

write to `flag1/2` and read of `flag2/1` are independent so can be reordered → error!

- We assume:
  - sequential consistency in absence of data race
  - for data races, values are limited by possible interleavings between synchronization points

Abstractions

- Based on Astrée for the sequential processes
- Takes scheduling into account
- OS entry points (semaphores, logbooks, sampling and queuing ports, buffers, blackboards, ...) are all stubbed (using Astrée stubbing directives)
- Interference between processes: flow-insensitive abstraction of the writes to shared memory and inter-process communications

Example of static analysis of a complex parallel application

- Degraded mode (5 processes, 100 000 LOCS)
  - 1h40 on 64-bit 2.66 GHz Intel server
  - 98 alarms
- Full mode (15 processes, 1 600 000 LOCS)
  - 50 h
  - 12 000 alarms !!! more work to be done !!! (e.g. analysis of complex data structures, logs, etc)

Conclusion

Cost-effective verification

- The *rumor* has it that:
  - Manuel validation (testing) is costly, unsafe, not a verification!
  - Formal proofs by theorem provers are extremely laborious hence costly
  - Model-checkers do not scale up
- Why not try abstract interpretation?
  - Domain-specific static analysis scales and can deliver no false alarm

Characteristics of ASTRÉE (cont'd)

Sound: – ASTRÉE is a bug eradicator: finds all bugs in a well-defined class (runtime errors)

– ASTRÉE is not a bug hunter: finding some bugs in a well-defined class (e.g. by *bug pattern detection* like FindBugs™, PREfast or PMD)

– ASTRÉE is exhaustive: covers the whole state space (≠ MAGIC, CBMC)

– ASTRÉE is comprehensive: never omits potential errors (≠ UNO, CMC from coverity.com) or sort most probable ones to avoid overwhelming messages (≠ Splint)

Characteristics of ASTRÉE (cont'd)

Static: compile time analysis (≠ run time analysis Rational Purify, Parasoft Insure++)

Program Analyzer: analyzes programs not micromodels of programs (≠ PROMELA in SPIN or Alloy in the Alloy Analyzer)

Automatic: no end-user intervention needed (≠ ESC Java, ESC Java 2), or PREfast (annotate functions with intended use)

Characteristics of ASTRÉE (cont'd)

Multiabstraction: uses many numerical/symbolic abstract domains (≠ symbolic constraints in Bane or the canonical abstraction of TVLA)

Infinitary: all abstractions use infinite abstract domains with widening/narrowing (≠ model checking based analyzers such as Bandera, Bogor, Java PathFinder, Spin, VeriSoft)

Efficient: always terminate (≠ counterexample-driven automatic abstraction refinement BLAST, SLAM)

Characteristics of ASTRÉE (cont'd)

Extensible/Specializable: can easily incorporate new abstractions (and reduction with already existing abstract domains) ($\neq$ general-purpose analyzers PolySpace Verifier)

Domain-Aware: knows about control/command (e.g. digital filters) (as opposed to specialization to a mere programming style in C Global Surveyor)

Parametric: the precision/cost can be tailored to user needs by options and directives in the code

Characteristics of ASTRÉE (cont'd)

Automatic Parametrization: the generation of parametric directives in the code can be programmed (to be specialized for a specific application domain)

Modular: an analyzer instance is built by selection of O-CAML modules from a collection each implementing an abstract domain

Precise: very few or no false alarm when adapted to an application domain $\longrightarrow$ it is a VERIFIER!

References

If you have only time to have a look at one recent reference

- J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival

  Static analysis and verification of aerospace software by abstract interpretation

  AAIA Infotech@Aerospace 2010, Atlanta, 20—22 April 2010, Georgia, AIAA 2010-3385

  `http://pdf.aiaa.org/preview/2010/CDReadyMIAA10_2358/PV2010_3385.pdf`

Basic introductions to abstract abstract interpretation

1. Patrick Cousot.
   Interprétation abstraite.
   Technique et Science Informatique, Vol. 19, Nb 1-2-3. Janvier 2000, Hermès, Paris, France. pp. 155—164.

2. Patrick Cousot.
   Abstract Interpretation Based Formal Methods and Future Challenges.
   In *Informatics, 10 Years Back - 10 Years Ahead*, R. Wilhelm (Ed.), Lecture Notes in Computer Science 2000, pp. 138—156, 2001.

3. Patrick Cousot & Radhia Cousot.
   Basic Concepts of Abstract Interpretation.
   In *Building the Information Society*, R. Jacquard (Ed.), Kluwer Academic Publishers, pp. 359—366, 2004.

Basic references on abstract interpretation

4. Patrick Cousot & Radhia Cousot.
   Static Determination of Dynamic Properties of Programs.
   In *Proceedings of the second international symposium on Programming*, B. Robinet (Ed), Paris, France, pages 106—130, 13—15 April 1976, Dunod, Paris.

5. Patrick Cousot & Radhia Cousot.
   Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.
   In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238—252, Los Angeles, California, 1977. ACM Press, New York.

6. Patrick Cousot & Radhia Cousot.
   Static determination of dynamic properties of recursive procedures.
   In *IFIP Conference on Formal Description of Programming Concepts*, E.J. Neuhold, (Ed.), pages 237—277, St-Andrews, N.B., Canada, 1977. North-Holland Publishing Company (1978).

7. Patrick Cousot & Radhia Cousot.
   Systematic Design of Program Analysis Frameworks.
   In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269—282, San Antonio, Texas, 1979. ACM Press, New York.

8. Patrick Cousot & Radhia Cousot.
   Abstract interpretation frameworks.
   Journal of Logic and Computation, 2(4):511—547, August 1992.

Basic references on abstract interpretation (cont'd)

9. Patrick Cousot & Radhia Cousot.
   Comparing the Galois connection and widening/narrowing approaches to abstract interpretation.
   Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92, Leuven, Belgium, 13—17 August 1992, Volume 631 of Lecture Notes in Computer Science, pages 269—295. © Springer-Verlag, Berlin, Germany, 1992.

10. Patrick Cousot.
    The Calculational Design of a Generic Abstract Interpreter.
    In Broy, M., and Steinbrüggen, R. (eds.): *Calculational System Design.* NATO ASI Series F. Amsterdam: IOS Press, 1999.

References on ASTRÉE

11. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival.
    Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter.
    In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen and D.A. Schmidt and I.H. Sudborough (Editors). Volume 2566 of Lecture Notes in Computer Science, pp. 85—108, © Springer.

12. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, & Xavier Rival.
    A Static Analyzer for Large Safety-Critical Software.
    In *PLDI 2003 — ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, 2003 Federated Computing Research Conference, June 7—14, 2003, San Diego, California, USA, pp. 196—207, © ACM.

13. Jérôme Feret.
    Static analysis of digital filters.
    In *ESOP 2004 — European Symposium on Programming*, D. Schmidt (editor), Mar. 27 —Apr. 4, 2004, Barcelona, ES, Volume 2986 of Lecture Notes in Computer Science, pp. 33—48, © Springer.

14. Laurent Mauborgne.
    ASTRÉE: verification of absence of run-time error.
    In *Building the Information Society*, R. Jacquard (Ed.), Kluwer Academic Publishers, pp. 385—392, 2004.

References on ASTRÉE (cont'd)

15. Antoine Miné.
Relational abstract domains for the detection of floating-point run-time errors.
In *ESOP 2004 — European Symposium on Programming*, D. Schmidt (editor), Mar. 27 — Apr. 4, 2004, Barcelona, Volume 2986 of Lecture Notes in Computer Science, pp. 3—17, © Springer.

16. Antoine Miné.
Weakly relational numerical abstract domains.
Thèse de l'École polytechnique, 6 December 2004.

17. Jérôme Feret.
The arithmetic-geometric progression abstract domain.
In *VMCAI 2005 — Verification, Model Checking and Abstract Interpretation*, R. Cousot (editor), Volume 3385 of Lecture Notes in Computer Science, pp. 42—58, 17—19 January 2005, Paris, © Springer.

18. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival.
The ASTRÉE analyser.
In *ESOP 2005 — The European Symposium on Programming*, M. Sagiv (editor), Volume 3444 of Lecture Notes in Computer Science, pp. 21—30, 2—10 April 2005, Edinburgh, © Springer.

References on ASTRÉE (cont'd)

19. Laurent Mauborgne & Xavier Rival.
Trace Partitioning in Abstract Interpretation Based Static Analyzer.
In *ESOP 2005 — ; The European Symposium on Programming*, M. Sagiv (editor), Volume 3444 of Lecture Notes in Computer Science, pp. 5—20, 2—10 April 2005, Edinburgh, © Springer.

20. Xavier Rival.
Understanding the Origin of Alarms in ASTRÉE.
In *SAS'05 — The 12th International Static Analysis Symposium*, Chris Hankin & Igor Siveroni (editors), Volume 3672 of Lecture Notes in Computer Science, pp. 303—319, 7—9 September 2005, London, UK, © Springer.

21. David Monniaux.
The Parallel Implementation of the Astree Static Analyzer.
In *APLAS 2005 — The Third Asian Symposium on Programming Languages and Systems*, Kwangkeun Yi (editor), Volume 3780 of Lecture Notes in Computer Science, pp. 86—96, 2—5 November 2005, Tsukuba, Japan, © Springer.

22. Xavier Rival.
Abstract Dependences for Alarm Diagnosis.
In *APLAS 2005 — The Third Asian Symposium on Programming Languages and Systems*, Kwangkeun Yi (editor), Volume 3780 of Lecture Notes in Computer Science, pp. 347—363, 2—5 November 2005, Tsukuba, Japan, © Springer.

References on ASTRÉE (cont'd)

23. Antoine Miné.
Symbolic Methods to Enhance the Precision of Numerical Abstract Domains.
In *VMCAI 2006 — Seventh International Conference on Verification, Model Checking and Abstract Interpretation*, E. Allen Emerson & Kedar S. Namjoshi (editors), Volume 3855 of Lecture Notes in Computer Science, pp. 348—363, 8—10 January 2006, Charleston, South Carolina, USA, © Springer.

24. Antoine Miné.
Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics.
In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference for Languages, Compilers, and Tools for Embedded Systems (LCTES 2006)*, 14—16 June 2006, Ottawa, Ontario, Canada. ACM Press, pp. 54—63.

25. Patrick Cousot.
L'analyseur statique ASTRÉE, Grand Colloque TIC 2006, Session RNTL « Systèmes embarqués », Centre de congrès, Lyon, 15 novembre 2006.

26. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, & Xavier Rival.
Combination of Abstractions in the ASTRÉE Static Analyzer. In *11th Annual Asian Computing Science Conference (ASIAN'06),* National Center of Sciences, Tokyo, Japan, December 6—8, 2006. LNCS 4435, Springer, Berlin, pp. 272—300, 2008.

References on ASTRÉE (cont'd)

27. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival.
Varieties of Static Analyzers: A Comparison with ASTRÉE, invited paper.
First IEEE & IFIP International Symposium on ``Theoretical Aspects of Software Engineering", TASE'07, Shanghai, China, 6—8 June 2007, pp. 3—17.

28. Patrick Cousot.
Proving the Absence of Run-Time Errors in Safety-Critical Avionics Code.
In *EMSOFT 2007, Embedded Systems Week,* Salzburg, Austria, September 30th, 2007, pp. 7—9, ACM Press.

29. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival.
Why does ASTRÉE scale up.
Formal Methods in System Design, Springer, to appear, 2010.

References on the industrial use of abstract interpretation

30. David Delmas and Jean Souyris.
ASTRÉE: from Research to Industry.
Proc. 14th International Static Analysis Symposium, SAS 2007, G. Filé & H. Riis-Nielson (eds), Kongens Lyngby, Denmark, 22-24 August 2007, LNCS 4634, pp. 437—451, © Springer, Berlin.

31. Jean Souyris and David Delmas.
Experimental Assessment of ASTRÉE on Safety-Critical Avionics Software.
Proc. Int. Conf. Computer Safety, Reliability, and Security, SAFECOMP 2007, Francesca Saglietti and Norbert Oster (Eds.), Nuremberg, Germany, September 18—21, 2007, Volume 4680 of Lecture Notes in Computer Science, pp. 479—490, © Springer, Berlin.

32. O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, M. Turin.
Space software validation using Abstract Interpretation.
Proc. 13thData Systems in Aerospace, DASIA 2009, Istanbul, Turkey, 26-29 May 2009, © Eurospace, Paris.

The End