« Software Verification by Abstract Interpretation and the ASTRÉE Static Analyzer »

Patrick Cousot
École normale supérieure
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr   www.di.ens.fr/~cousot

Computer Science Department — Stony Brook University
18 January 2008

Abstract

Abstract interpretation is a theory of sound approximation of the behavior of dynamic systems, in particular the semantics of programming languages. This is the formal basis for automatic correctness proofs by static analysers considering an over-approximation of the set of all possible executions of the program. Contrary to bug-finding methods (e.g. by test, bounded model-checking or error pattern search), no potential error is ever omitted. Hence the proof of satisfaction of a specification is always mathematically valid. Contrary to refinement-based methods, termination is always guaranteed. However, by undecidability of such proofs, the abstraction may yield false alarms whenever a synthesized inductive argument (e.g. a loop invariant) is too weak to make the proof. In this case, some executions considered in the abstract, that is in the over-approximation, might lead to an error while not corresponding to a concrete, that is actual, execution. All the difficulty of the undecidable verification problem is therefore to design safe/sound over-approximations that are coarse enough to be effectively computable by the static analyzer and precise enough to avoid false alarms (the errors leading to true alarms can only be eliminated by correcting the program that does not satisfy the specification).

After a brief introduction to abstract interpretation, we will present the ASTRÉE static analyser (`www.astree.ens.fr`) for proving the absence of runtime errors (such as buffer overrun, dangling pointer, division by zero, float overflow, modular integer arithmetic overflow, ...) in real-time synchronous control/command C applications. The ASTRÉE static analyser uses generalist abstractions (like intervals, octagons, decision trees, symbolic execution, etc) and abstractions for the specific application domain (to cope with filters, integrators, slow divergences due to rounding errors, etc). Since 2003, these domain-specific abstractions allowed for the verification of the absence of runtime errors in several large avionic software, a world première.

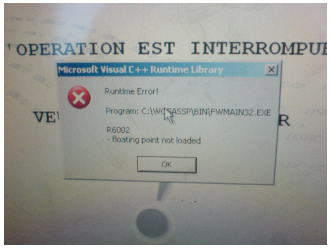
Contents

---

1. Motivation

Bugs Now Show-Up in Everyday Life

- Bugs now appear frequently in everyday life (banks, cars, telephones, . . .)
- Example (HSBC bank ATM[1] at 19 Boulevard Sébastopol in Paris, failure on Nov. 21st 2006 at 8:30 am):


'OPERATION EST INTERROMPUE

Microsoft Visual C++ Runtime Library

Runtime Error!

Program: C:\WCSASSP\BIN\FWMAIN32.EXE

R6002
- floating point not loaded

---

[1] cash machine, cash dispenser, automatic teller machine.

A Strong Need for Software Better Quality

- Poor software quality is not acceptable in safety and mission critical software applications.

- The present state of the art in software engineering does not offer sufficient quality garantees

---

The Complexity of Software Design

- The design of complex software is difficult and economically critical
- Example (`www.designnews.com/article/CA6475332.html`):

“Boeing Confirms 787 Delay, Fasteners, Flight Control Software Code Blamed
John Dodge, Editor-in-Chief – Design News, September 5, 2007

> Boeing officials confirmed today that a fastener shortage and problems with flight control software have pushed “first flight” of the Boeing 787 Dreamliner to sometime between mid-November and mid-December[2].
>
> ...
>
> The software delays involve Honeywell Aerospace, which is responsible for flight control software. The work on this part of the 787 was simply underestimated, said Bair.”

---

[2] Bill Rigby of Reuters announced that Boeing delays 787 by 3 months on Wed Jan 16, 2008 12:37pm EST.

The Security of Complex Software

- Complex software is subject to security vulnerabilies
- Example (`www.wired.com/politics/security/news/2008/01/dreamliner_security`)

“FAA: Boeing’s New 787 May Be Vulnerable to Hacker Attack
Kim Zetter, freelance journalist in Oakland, CA, Jan. 4, 2008

> Boeing’s new 787 Dreamliner passenger jet may have a serious security vulnerability in its onboard computer networks ...
>
> According to the FAA document published in the Federal Register (mirrored at Cryptome.org), the vulnerability exists because the plane’s computer systems connect the passenger network with the flight-safety, control and navigation network. It also connects to the airline’s business and administrative-support network, which communicates maintenance issues to ground crews.

Tool-Based Software Design Methods

- New tool-based software design methods will have to emerge to face the unprecedented growth and complexification of critical software
- E.g. FCPC (Flight Control Primary Computer)
  - A220: 20 000 LOCs,
  - A340 (V1): 130 000 LOCS
  - A340 (V2): 250 000 LOCS
  - A380: 1.000.000 LOCS
  - A350: static analysis to be integrated in the software production

Validation/Formal Methods

- Bug-finding methods : unit, integration, and system testing, dynamic verification, bounded model-checking, error pattern mining, . . .
- Absence of bug proving methods : formally prove that the semantics of a program satisfies a specification
  - theorem-proving & proof checking
  - model-checking
  - static analysis
- In practice : complementary methods are used, very difficult to scale up

Static Analysis

A *static analyzer* is a program that

- takes as input:
  - a program P (written in some given programming language $\mathbb{P}$ with a given semantics $\mathfrak{S}_{\mathbb{P}}$)
  - a specification S (implicit $S\llbracket P\rrbracket$ or written in some specification language $\mathbb{S}$ with a given semantics $\mathfrak{S}_{\mathbb{S}}$)
- *always terminates* and delivers *automatically* as output:
  - a diagnosis on the validity of the program semantics with respect the specification semantics

Difficulties of Static Analysis

- automatic + infinite state + termination $\Longrightarrow$ undecidable!
- for a programming (and a specification) language, not for a given model of a given program:

$$\forall P \in \mathbb{P} : \forall S \in \mathbb{S} : \mathfrak{S}_{\mathbb{P}}\llbracket P\rrbracket \subseteq \mathcal{S}_{\mathbb{S}}\llbracket P, S\rrbracket ?$$

or, more simply for an *implicit specification* $\mathfrak{S}\llbracket P\rrbracket$:

$$\forall P \in \mathbb{P} : \mathfrak{S}_{\mathbb{P}}\llbracket P\rrbracket \subseteq \mathfrak{S}\llbracket P\rrbracket ?$$

Soundness and Completeness

- Soundness: for all $P \in \mathbb{P}$, if the answer is yes (no) then $\mathfrak{S}_{\mathbb{P}}[\![P]\!] \subseteq \mathfrak{S}[\![P]\!]$ (resp. $\mathfrak{S}_{\mathbb{P}}[\![P]\!] \not\subseteq \mathfrak{S}[\![P]\!]$)
- Completeness: for all $P \in \mathbb{P}$, if $\mathfrak{S}_{\mathbb{P}}[\![P]\!] \subseteq \mathfrak{S}[\![P]\!]$ ($\mathfrak{S}_{\mathbb{P}}[\![P]\!] \not\subseteq \mathfrak{S}[\![P]\!]$) then the answer is yes (resp. no)

We always require SOUNDNESS!

Undecidability $\Longrightarrow$ no completeness

Problems with Formal Methods

- Formal specifications (abstract machines, temporal logic, . . .) are costly, complex, error-prone, difficult to maintain, not mastered by casual programmers
- Formal semantics of the specification and programming language are inexistant, informal, irrealistic or complex
- Formal proofs are partial (static analysis), do not scale up (model checking) or need human assistance (theorem proving & proof assistants)

  ⇒ High costs (for specification, proof assistance, etc).

Avantages of Static Analysis

- Formal specifications are implicit (no need for explicit, user-provided specifications)
- Formal semantics are approximated by the static analyzer (no user-provided models of the program)
- Formal proofs are automatic (no required user-interaction)
- Costs are low (no modification of the software production methodology)
- Scales up to 100.000 to 1.000.000 LOCS
- Rapid and large diffusion in embedded software production industries

---

Disadvantages of Static Analysis

- Imprecision (acceptable in some applications like WCET or program optimization)
- Incomplete for program verification
- False alarms are due to unsuccessful automatic proofs in 5 to 15% of the cases

For example, 1% of 500.000 potential (true or false) alarms is 5.000, too much to be handled by hand!

Remedies to False Alarms in ASTRÉE

- ASTRÉE is specialized to specific program properties [3]
- ASTRÉE is specialized to real-time synchronous control/command programs written in C
- ASTRÉE offers possibilities of refinement [4]

The cost of adapting ASTRÉE to a specific program, should be a small fraction of the cost to test the specific program properties verified by ASTRÉE.

---

[3] proof of absence of runtime errors

[4] parametrizations and analysis directives

---

2. Informal Introduction to Abstract Interpretation

Abstract Interpretation

There are two fundamental concepts in computer science (and in sciences in general) :

- **Abstraction** : to reason on complex systems
- **Approximation** : to make effective undecidable computations

These concepts are formalized by abstract interpretation [CC77, Cou78, CC79, Cou81, CC92a]

References

[POPL '77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM POPL*.

[Thesis '78] P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse ès sci. math. Grenoble, march 1978.

[POPL '79] P. Cousot & R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *ACM POPL*.

Applications of Abstract Interpretation

- Static Program Analysis [CC77], [CH78], [CC79] including Dataflow Analysis; [CC79], [CC00], Set-based Analysis [CC95], Predicate Abstraction [Cou03], . . .
- Grammar Analysis and Parsing [CC03];
- Hierarchies of Semantics and Proof Methods [CC92b], [Cou02];
- Typing & Type Inference [Cou97];
- (Abstract) Model Checking [CC00];
- Program Transformation (including program optimization, partial evaluation, etc) [CC02];

Applications of Abstract Interpretation (Cont'd)

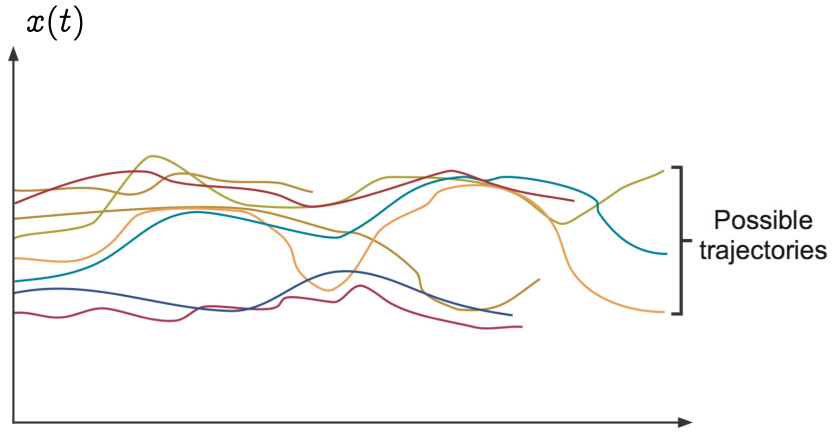
- Software Watermarking [CC04];
- Bisimulations [RT04, RT06];
- Language-based security [GM04];
- Semantics-based obfuscated malware detection [PCJD07].
- Databases [AGM93, BPC01, BS97]
- Computational biology [Dan07]
- Quantum computing [JP06, Per06]

All these techniques involve sound approximations that can be formalized by abstract interpretation

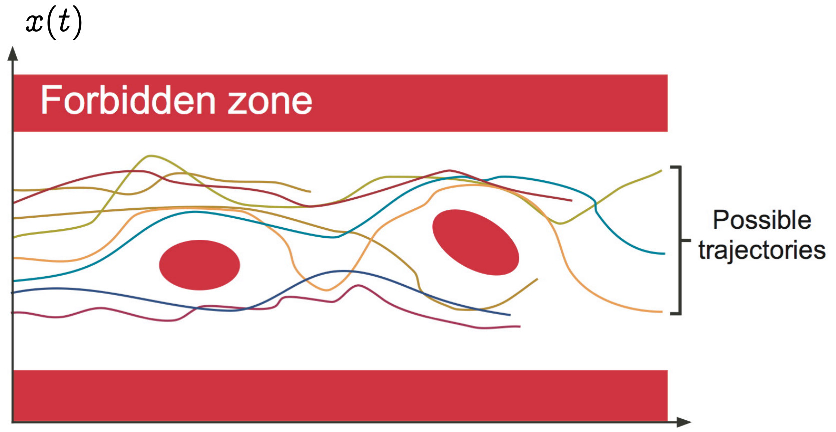
---

Principle of Abstraction

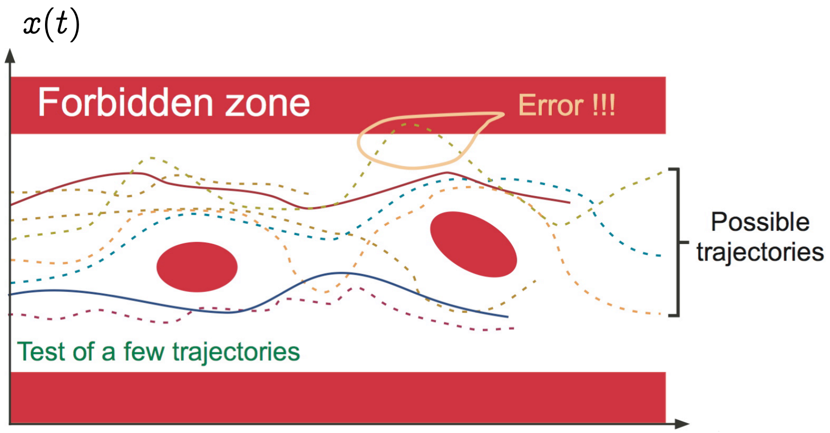
Operational semantics



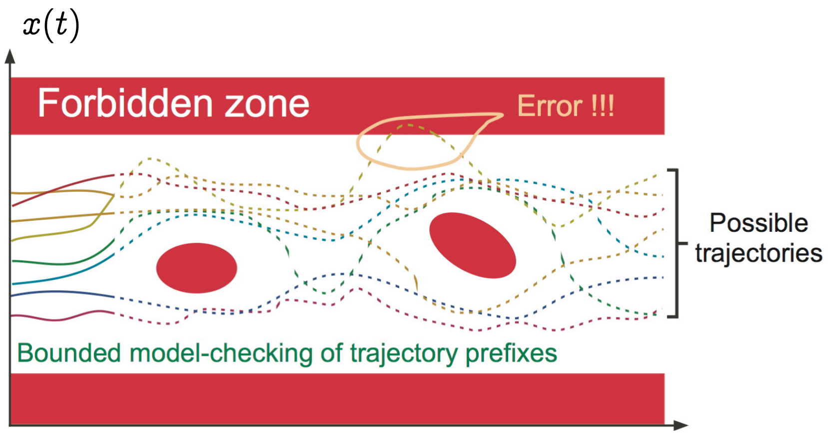
Safety property



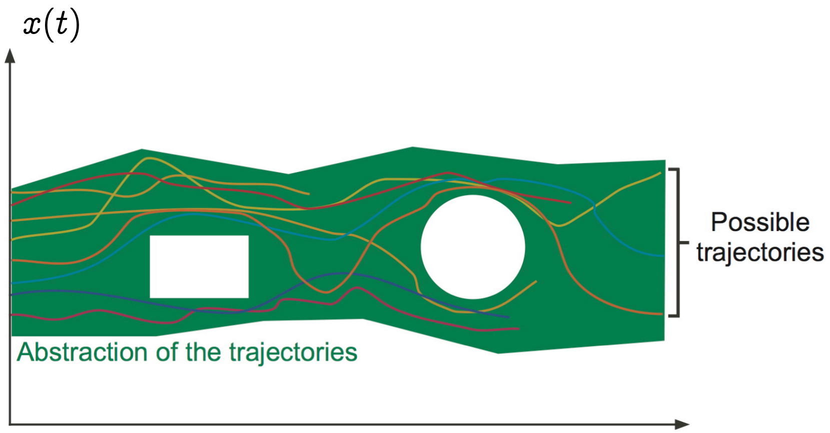
Test/Debugging is Unsafe



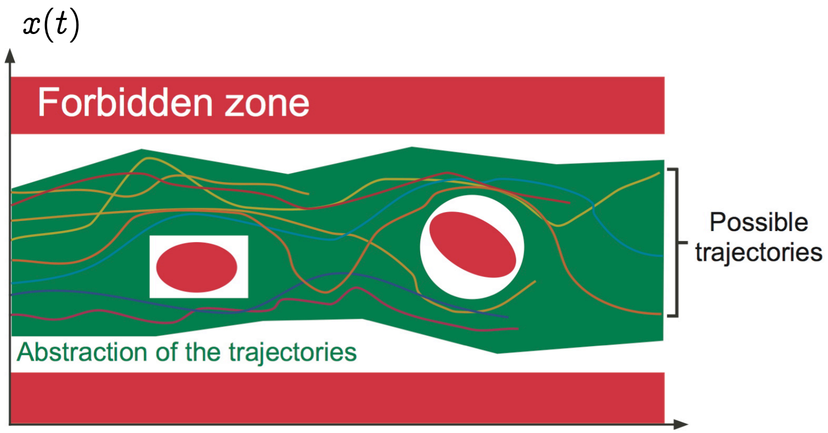
Bounded Model Checking is Unsafe



Over-Approximation

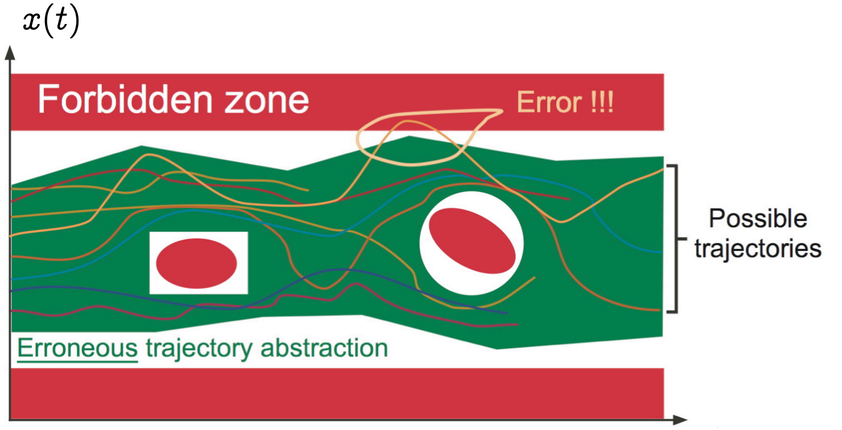


Abstract Interpretation is Sound



Soundness and Incompleteness

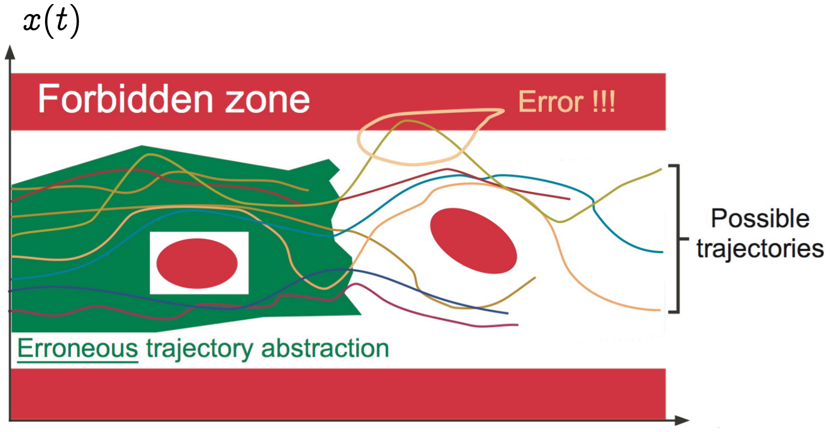
Soundness Requirement: Erroneous Abstraction[5]



---

[5] This situation is always excluded in static analysis by abstract interpretation.

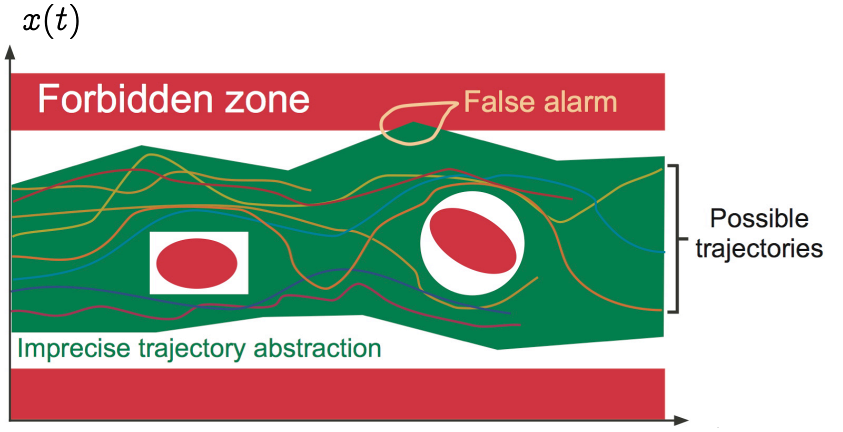
Soundness Requirement: Erroneous Abstraction[6]



---

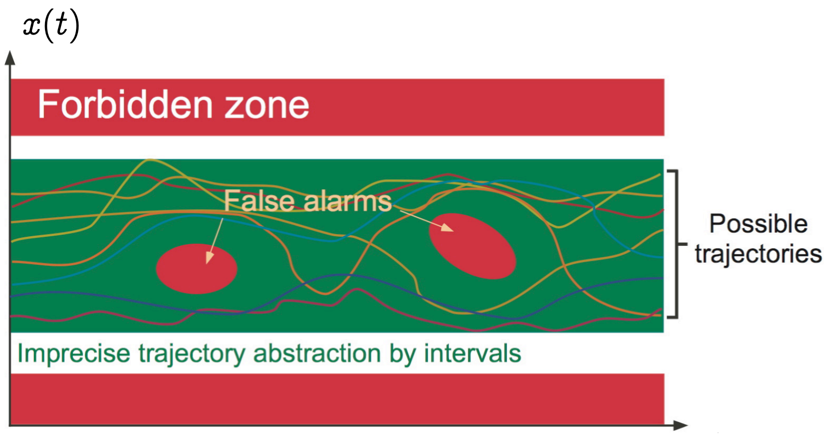
[6] This situation is always excluded in static analysis by abstract interpretation.

Imprecision ⇒ False Alarms

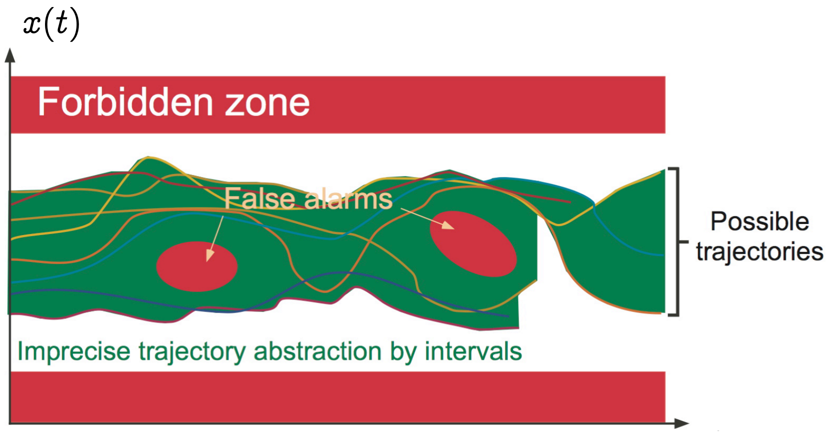


Design by Refinement

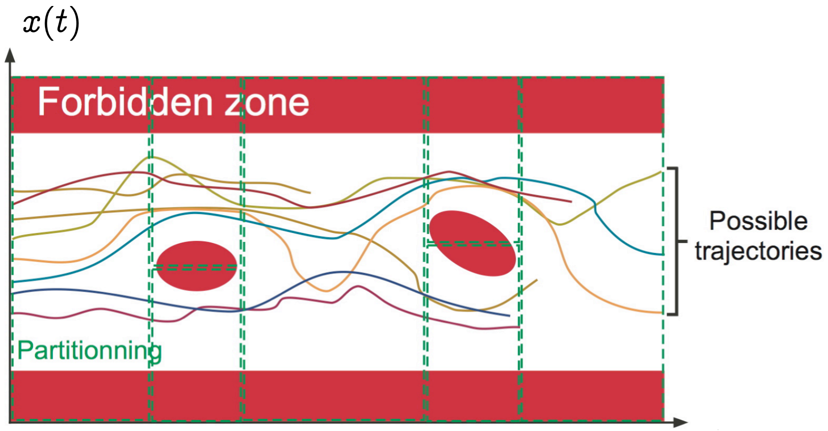
Global Interval Abstraction → False Alarms



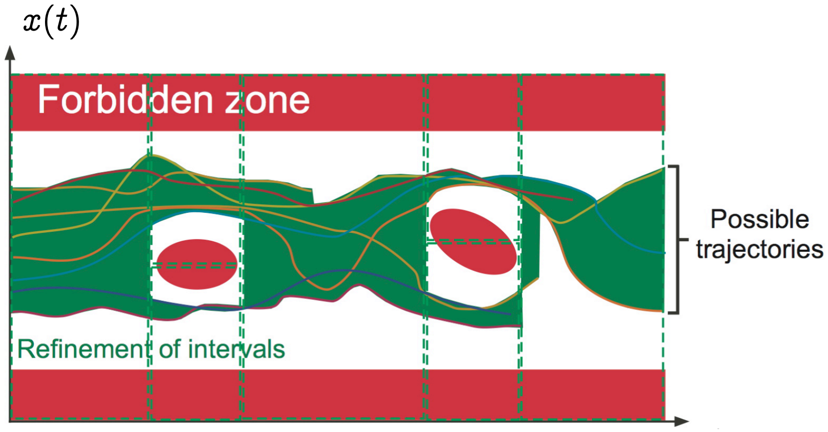
Local Interval Abstraction → False Alarms



Refinement by Partitionning

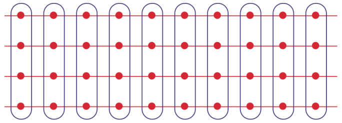


Intervals with Partitionning

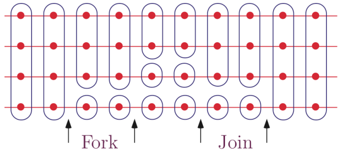


State-based versus Trace-based Partitionning

State-based partitionning at control points:



Trace-based partitionning at control points:



Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).

Trace Partitioning

Principle:

- Semantic equivalence:

```
if (B) { C1 } else { C2 }; C3
              ⇓
if (B) { C1; C3 } else { C2; C3 };
```

- More precise in the abstract: concrete execution paths are merged later.

Application:

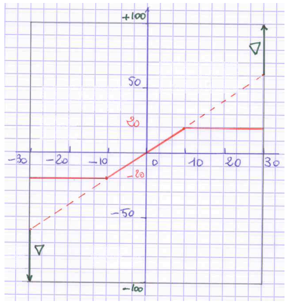
```
if (B)
  { X=0; Y=1; }
else
  { X=1; Y=0; }
R = 1 / (X-Y);
```

cannot result in a division by zero

Case analysis with loop unrolling

– Code Sample:

```
/* trace_partitionning.c */
void main() {
  float t[5] = {-10.0, -10.0, 0.0, 10.0, 10.0};
  float c[4] = {0.0, 2.0, 2.0, 0.0};
  float d[4] = {-20.0, -20.0, 0.0, 20.0};
  float x, r;
  int i = 0;
  __ASTREE_known_fact(((-30.0 <= x) && (x <= 30.0)));
  while ((i < 3) && (x >= t[i+1])) {
    i = i + 1;
  }
  r = (x - t[i]) * c[i] + d[i];
  __ASTREE_log_vars((r));
}
```



```
% astree -exec-fn main -no-trace -no-relational trace-partitioning.c |& egrep "(WARN)|(r in)"
direct = <float-interval: r in [-20, 20] >
%
% astree -exec-fn main -no-partition -no-trace -no-relational trace-partitioning.c \
  |& egrep "(WARN)|(r in)"
direct = <float-interval: r in [-100, 100] >
%
```

3. The ASTRÉE static analyzer

http://www.astree.ens.fr/

Project Members


Bruno BLANCHET[7]


Patrick COUSOT


Radhia COUSOT


Jérôme FERET


Laurent MAUBORGNE


Antoine MINÉ


David MONNIAUX[8]


Xavier RIVAL

---

[7] Nov. 2001 — Nov. 2003.

[8] Nov. 2001 — Aug. 2007.

Programs Analyzed by ASTRÉE and their Semantics

Programs analysed by ASTRÉE

- Application Domain: large safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.
- C programs:
  - with
    - basic numeric datatypes, structures and arrays
    - pointers (including on functions),
    - floating point computations
    - tests, loops and function calls
    - limited branching (forward `goto`, `break`, `continue`)

- with (cont'd) **NEW**
  - `union` [Min06a]
  - pointer arithmetics & casts [Min06a]
- without
  - dynamic memory allocation
  - recursive function calls
  - unstructured/backward branching
  - conflicting side effects
  - C libraries, system calls (parallelism)

Such limitations are quite common for embedded safety-critical software.

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 [FHL+01].

Concrete Operational Semantics

- International norm of C (ISO/IEC 9899:1999)
- *restricted by* implementation-specific behaviors depending upon the machine and compiler (e.g. representation and size of integers, IEEE 754-1985 norm for floats and doubles)
- *restricted by* user-defined programming guidelines (such as no modular arithmetic for signed integers, even though this might be the hardware choice)
- *restricted by* program specific user requirements (e.g. `assert`, execution stops on first runtime error[9])

---

[9] semantics of C unclear after an error, equivalent if no alarm

---

The Semantics of C is Hard (Ex. 1: Floats)

"*Put* `x` *in* $[\mathtt{m}, \mathtt{M}]$ *modulo* $(\mathtt{M} - \mathtt{m})$":

```
x' = x - (int) ((x-m)/(M-m))*(M-m);
```

- The programmer thinks $\mathtt{x'} \in [\mathtt{m}, \mathtt{M}]$
- But with $\mathtt{M} = 4095$, $\mathtt{m} = -\mathtt{M}$, IEEE double precision, and `x` is the greatest float strictly less than `M`, then $\mathtt{x'} = \mathtt{m} - \epsilon$ (ϵ very small).

Floats are not real.

The Semantics of C is Hard (Ex. 2: Runtime Errors)

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

Execution stops after a runtime error with unpredictable results[10].

---

[10] Equivalent semantics if no alarm.

Different Classes of Run-time Errors

1. Errors terminating the execution[11]. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.
2. Errors not terminating the execution with predictable outcome[12]. ASTRÉE warns and continues with worst-case assumptions.
3. Errors not terminating the execution with unpredictable outcome[13]. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.

⇒ ASTRÉE is sound with respect to C standard, unsound with respect to C implementation, unless no false alarm.

---

[11] floating-point exceptions e.g. (invalid operations, overflows, etc.) when traps are activated

[12] e.g. overflows over signed integers resulting in some signed integer.

[13] e.g. memory corruptionss.

Specification Proved by ASTRÉE

Implicit Specification: Absence of Runtime Errors

- No violation of the norm of C (e.g. array index out of bounds, division by zero)
- No implementation-specific undefined behaviors (e.g. maximum short integer is 32767, NaN)
- No violation of the programming guidelines (e.g. static variables cannot be assumed to be initialized to 0)
- No violation of the programmer assertions (must all be statically verified).

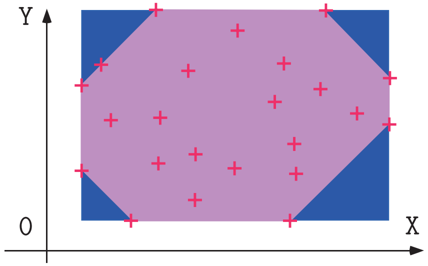
General Purpose Abstractions

Abstraction is Extremely Hard

- The analysis must be automatic (no user interaction)
- The abstraction must
  - ensure termination (and efficiency) of the analysis
  - be sound (Astrée is a verifier, not a bug-finder)
  - scale up (100.000 to 1.000.000 LOCs)
  - be precise (no false alarm)

A grand challenge

General-Purpose Abstract Domains: Intervals and Octagons



Intervals:

$$\begin{cases} 1 \leq x \leq 9 \\ 1 \leq y \leq 20 \end{cases}$$

Octagons [Min01]:

$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 77 \\ 1 \leq y \leq 20 \\ x - y \leq 04 \end{cases}$$

Difficulties: many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program and analyzer) [CC77, Min01, Min04a]

Termination

SLAM uses CEGAR and does not terminate[14] on

```
% cat slam.c
int main() { int x, y;
  x = 0; y = 0;
  while (x < 2147483647)
     { x = x + 1; y = y + 1; }
  __ASTREE_assert((x == y));
}
```

whereas Astrée uses widening/narrowing-based extrapolation techniques to prove the assertion

```
% astree -exec-fn main slam.c |& egrep "WARN"
%
```

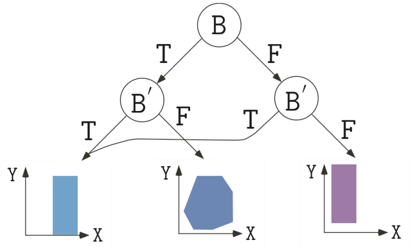
---

[14] CEGAR cannot generate the invariant `y = x - 1` so produces all counter examples $x = i + 1 \wedge y = i$, $i = 0, 1, 2, 3, \ldots$

Boolean Relations for Boolean Control

– Code Sample:

```
/* 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 leafs

---

Modular Arithmetic

Modular arithmetics is not very intuitive

In C:

```
% 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
```

Static Analysis with 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 unkown value.

Float Overflow

Float Arithmetics does Overflow

In C:

```
% 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]
```

The Ariane 5.01 maiden flight

- June 4th, 1996 was the maiden flight of Ariane 5

The Ariane 5.01 maiden flight failure

- June 4th, 1996 was the maiden flight of Ariane 5
- The launcher was detroyed after 40 seconds of flight because of a software overflow[15]

---

[15] A 16 bit piece of code of Ariane 4 had been reused within the new 32 bit code for Ariane 5. This caused an uncaught overflow, making the launcher uncontrolable.

Rounding

Example of rounding error

```
/* 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
```

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

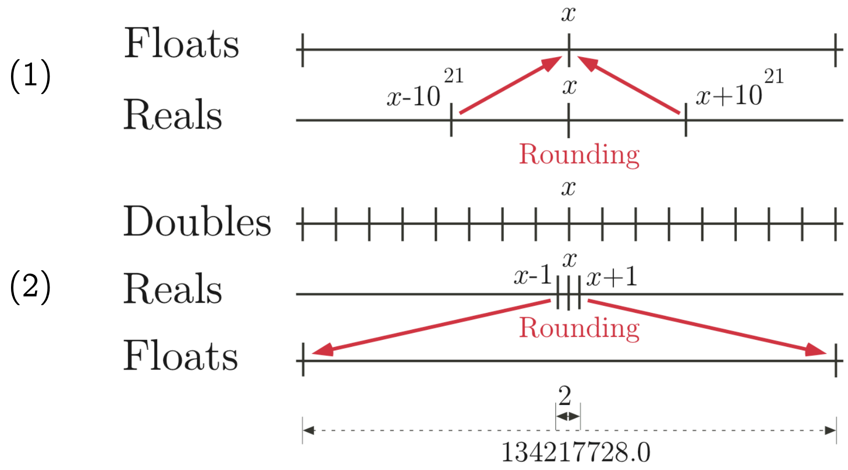
Example of rounding error

```
/* 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



Static analysis with ASTRÉE[16]

```
% 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] >
```

---

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

Example of accumulation of small 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 $(0.1)_{10} = (0.0001100110011001100\ldots)_2$

Static analysis with 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
```

The Patriot missile failure

- "On February 25th, 1991, a Patriot missile … failed to track and intercept an incoming Scud (*)."
- The software failure was due to accumulated rounding error (†)

(*) This Scud subsequently hit an Army barracks, killing 28 Americans.

(†)
- "Time is kept continuously by the system's internal clock in tenths of seconds"
- "The system had been in operation for over 100 consecutive hours"
- "Because the system had been on so long, the resulting inaccuracy in the time calculation caused the range gate to shift so much that the system could not track the incoming Scud"

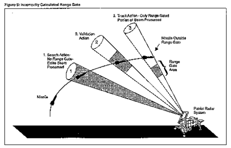


Figure 5: Incorrectly Calculated Range Gate

Scaling

Static Analysis of Scaling with ASTRÉE

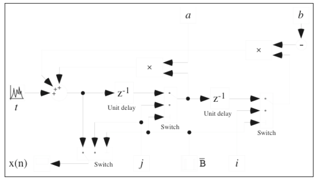
```
% cat -n scale.c                    % gcc scale.c
 1 int main () {                    % ./a.out
 2  float x; x = 0.70000001;        x = 0.699999988079071
 3  while (1) {
 4   x = x / 3.0;
 5   x = x * 3.0;
 6   __ASTREE_log_vars((x));
 7   __ASTREE_wait_for_clock(());
 8  }
 9 }
```

```
% 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] >
%
```

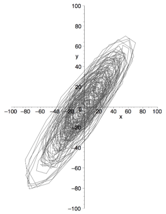
Filtering

Ellipsoid Abstract Domain for Filters

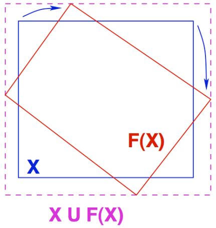
2^{d} Order Digital Filter:



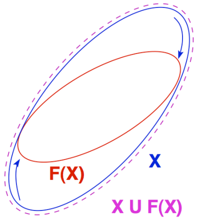
- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.



execution trace



unstable interval



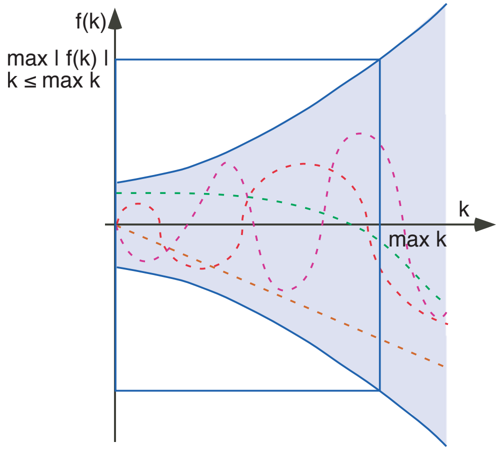
stable ellipsoid

Filter Example [Fer04]

```
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; }
}
```

Time Dependence

Overapproximation with an Arithmetic-Geometric Progression



Arithmetic-geometric progressions[17] [Fer05]

- Abstract domain: $(R^+)^5$
- Concretization:

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

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

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

i.e. any function bounded by the arithmetic-geometric progression[18].

---

[17] here in R, in practice in floats, so rounding must be taken into account [].

[18] Note that exhaustive enumeration would be simply hopeless.

Example 1: Bounding Increments [Fer05]

```
% 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 2: Accumulation of Small 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 $(0.1)_{10} = (0.0001100110011001100\ldots)_2$

Static Analysis with ASTRÉE

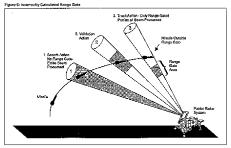
```
% cat -n rounding.c
     1  int main () {
     2   double x; x = 0.0;
     3   while (1) {
     4    x = x + 1.0/10.0;
     5    __ASTREE_log_vars((x));
     6    __ASTREE_wait_for_clock(());
     7   }
     8  }
% cat rounding.config
 __ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem 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
```

The Patriot missile failure

- “On February 25th, 1991, a Patriot missile … failed to track and intercept an incoming Scud (*).”
- The software failure was due to accumulated rounding error (†)

(*) This Scud subsequently hit an Army barracks, killing 28 Americans.

(†) – “Time is kept continuously by the system’s internal clock in tenths of seconds”
- “The system had been in operation for over 100 consecutive hours”
- “Because the system had been on so long, the resulting inaccuracy in the time calculation caused the range gate to shift so much that the system could not track the incoming Scud”



Example 3: Time Dependent Deviations [Fer05]

```
% 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
```

Incompleteness

Astrée does not know that

$$\forall x, y \in \mathbb{Z} : 7y^2 - 1 \neq x^2$$

so on the following program

```
void main() { int x, y;
  if ((-4681 < y) && (y < 4681) && (x < 32767) && (-32767 < x) && ((7*y*y - 1) == x*x))
     { y = 1 / x; };
}
```

it produces a false alarm

```
% astree -exec-fn main false-alarm.c |& egrep "WARN"
false-alarm.c:5.9-14::[call#main@1:]: WARN: integer division by zero ([-32766, 32766]
and {1} / Z)
%
```

Zero False Alarm Objective

Industrial constraints require Astrée to be extremely precise:

- Astrée is designed for a well-identified family of programs
- The analysis can be tuned using
  - parameters
  - analysis directives (which insertion can be automated)
  - extensions of the analyzer (by the tool designers)

Example of directive

```
% cat repeat1.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;

  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
% astree -exec-fn main repeat1.c |& egrep "WARN"
repeat1.c:5.8-13::[call#main@2:loop@4>=4:]: WARN: signed int arithmetic
range [-2147483649, 2147483646] not included in [-2147483648, 2147483647]
%
```

Example of directive (Cont'd)

```
% cat repeat2.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;
  __ASTREE_boolean_pack((b,x));
  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
% astree -exec-fn main repeat2.c |& egrep "WARN"
%
```

The insertion of this directive could have been automated in ASTRÉE.

---

4. The industrial use of ASTRÉE

References

[1] D. Delmas and J. Souyris. ASTRÉE: *from Research to Industry*. Proc. 14th Int. Symp. SAS '07, G. Filé and H. Riis-Nielson (eds), 22–24 Aug. 2007, Kongens Lyngby, DK, LNCS 4634, pp. 437–451, Springer.

Example application

- Primary flight control software of the Airbus A340 family/A380 fly-by-wire system

- C program, automatically generated from a proprietary high-level specification (à la Simulink/SCADE)
- A340 family: 132,000 lines, 75,000 LOCs after preprocessing, 10,000 global variables, over 21,000 after expansion of small arrays, now × 2
- A380: × 3/7

Digital Fly-by-Wire Avionics[19]

---

[19] The electrical flight control system is placed between the pilot's controls (sidesticks, rudder pedals) and the control surfaces of the aircraft, whose movement they control and monitor.

Benchmarks (Airbus A340 Primary Flight Control Software)

- V1 [20], 132,000 lines, 75,000 LOCs after preprocessing
- Comparative results (commercial software):

  4,200 (false?) alarms, 3.5 days;
- Our results:

  $\underline{\underline{0}}$ alarms,

  40mn on 2.8 GHz PC, 300 Megabytes

  ⟶ A world première in Nov. 2003!

---

[20] “Flight Control and Guidance Unit” (FCGU) running on the “Flight Control Primary Computers” (FCPC). The three primary computers (FCPC) and two secondary computers (FCSC) which form the A340 and A330 electrical flight control system are placed between the pilot’s controls (sidesticks, rudder pedals) and the control surfaces of the aircraft, whose movement they control and monitor.

The main loop invariant for the A340 V1

A textual file over 4.5 Mb with

- 6,900 boolean interval assertions ($x \in [0; 1]$)
- 9,600 interval assertions ($x \in [a; b]$)
- 25,400 clock assertions ($x + \mathrm{clk} \in [a; b] \wedge x - \mathrm{clk} \in [a; b]$)
- 19,100 additive octagonal assertions ($a \leq x + y \leq b$)
- 19,200 subtractive octagonal assertions ($a \leq x - y \leq b$)
- 100 decision trees
- 60 ellipse invariants, etc …

involving over 16,000 floating point constants (only 550 appearing in the program text) × 75,000 LOCs.

(Airbus A380 Primary Flight Control Software)

- $\underline{\underline{0}}$ alarms (Nov. 2004), after some additional parametrization and simple abstract domains developments
- Now at 1,000,000 lines!

  34h,

  8 Gigabyte

  ⟶ A world grand première!

Possible origins of imprecision and how to fix it

In case of false alarm, the imprecision can come from:

- Abstract transformers (not best possible) ⟶ improve algorithm;
- Automatized parametrization (e.g. variable packing) ⟶ improve pattern-matched program schemata;
- Iteration strategy for fixpoints ⟶ fix widening [21];
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract ⟶ add a new abstract domain to the reduced product (e.g. filters).

---

[21] This can be very hard since at the limit only a precise infinite iteration might be able to compute the proper abstract invariant. In that case, it might be better to design a more refined abstract domain.

5. Conclusion

Characteristics of the ASTRÉE Analyzer

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 the ASTRÉE Analyzer (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 the ASTRÉE Analyzer (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 the Astrée Analyzer (Cont'd)

Extensible/Specializable: can easily incorporate new abstractions (and reduction with already existing abstract domains) (≠ 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 the Astrée Analyzer (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 ⟶ it is a VERIFIER!

The Future of the Astrée Analyzer

- Astrée has shown usable and useful in one industrial context (*electric flight control*):
  - as a R & D tool for A340 V2 and A380,
  - as a production tool for the A350;
- More applications are forthcoming (ES_PASS project);
- Industrialization is simultaneously under consideration.

The Future of Abstract Interpretation

- Abstract interpretation is
  - a theory
  - with effective applications
  - and unprecedented industrial accomplishments.
- Further investigations of the theory are needed (while its scope of application broaden)
- The demand for applications is quasi-illimited

THE END, THANK YOU

---

6. Bibliography

[AGM93] G. Amato, F. Giannotti, and G. Mainetto. Data sharing analysis for a database programming language via abstract interpretation. In R. Agrawal, S. Baker, and D.A.Bell, editors, *Proceedings of the Ninthteenth International Conference on Very Large Data Bases*, pages 405–415, Dublin, Ireland, 24–27 August 1993. MORGANKAUFMANN.

[BCC^{+}02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science 2566, pages 85–108. Springer, Berlin, Germany, 2002.

[BCC^{+}03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN '2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, San Diego, California, United States, 7–14 June 2003. ACM Press, New York, New York, United States.

[BPC01] J. Bailey, A. Poulovassilis, and C. Courtenage. Optimising active database rules by partial evaluation and abstract interpretation. In *Proceedings of the Eight International Workshop on Database Programming Languages*, Lecture Notes in Computer Science 2397, pages 300–317, Frascati, Italy, 8–10 september 2001. Springer, Berlin, Germany.

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In M. Aksit and S. Matsuoka, editors, *Proceedings of the Eleventh European Conference on Object-Oriented Programming, ECOOP '97*, Lecture Notes in Computer Science 1241. Springer, Berlin, Germany, Jyväskylä, Finland, 9–13 June 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, New York, United States.

[CC79] P. Cousot and R. 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, New York, United States.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC92b] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, United States, 1992. ACM Press, New York, New York, United States.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, United States, 25–28 June 1995. ACM Press, New York, New York, United States.

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Massachusetts, United States, January 2000. ACM Press, New York, New York, United States.

[CC02] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, United States, January 2002. ACM Press, New York, New York, United States.

[CC03] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theoretical Computer Science*, 290(1):531–544, January 2003.

[CC04] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, 14–16 January 2004. ACM Press, New York, New York, United States.

[CCF^{+}05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In M. Sagiv, editor, *Proceedings of the Fourteenth European Symposium on Programming Languages and Systems, ESOP '2005, Edinburg, Scotland*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, Berlin, Germany, 2–10 April 2005.

[CCF+06] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer, invited paper. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference, ASIAN 06*, Tokyo, Japan, 6–8 December 2006. Lecture Notes in Computer Science , Springer, Berlin, Germany. To appear.

[CCF+07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée, invited paper. In M. Hinchey, He Jifeng, and J. Sanders, editors, *Proceedings of the First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE '07*, pages 3–17, Shanghai, China, 6–8 June 2007. IEEE Computer Society Press, Los Alamitos, California, United States.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, New York, United States.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 March 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis, invited chapter. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, United States, 1981.

[Cou97] P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, New York, United States.

[Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1—2):47–103, 2002.

[Cou03] P. Cousot. Verification by abstract interpretation, invited chapter. In N. Dershowitz, editor, *Proceedings of the International Symposium on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268. Lecture Notes in Computer Science 2772, Springer, Berlin, Germany, Taormina, Italy, 29 June – 4 July 2003.

[Cou07] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code, invited tutorial. In *Proceedings of the Seventh ACM & IEEE International Conference on Embedded Software, EMSOFT '2007*, pages 7–9. ACM Press, New York, New York, United States, 2007.

[Dan07] V. Danos. Abstract views on biological signaling. In *Mathematical Foundations of Programming Semantics, Twentythird Annual Conference (MFPS XXIII)*, 2007.

[DS07] D. Delmas and J. Souyris. Astrée: from research to industry. In G. Filé and H. Riis-Nielson, editors, *Proceedings of the Fourteenth International Symposium on Static Analysis, SAS '07*, Kongens Lyngby, Denmark, Lecture Notes in Computer Science 4634, pages 437–451. Springer, Berlin, Germany, 22–24 August 2007.

[Fer04] J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proceedings of the Thirteenth European Symposium on Programming Languages and Systems, ESOP '2004, Barcelona, Spain*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, Berlin, Germany, March 27 – April 4, 2004.

[Fer05] J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Proceedings of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005)*, pages 42–58, Paris, France, 17–19 January 2005. Lecture Notes in Computer Science 3385, Springer, Berlin, Germany.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T.A. Henzinger and C.M. Kirsch, editors, *Proceedings of the First International Workshop on Embedded Software, EMSOFT '2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, Berlin, Germany, 2001.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, Venice, Italy, 2004. ACM Press, New York, New York, United States.

[JP06] Ph. Jorrand and S. Perdrix. Towards a quantum calculus. In *Proceedings of the Fourth International Workshop on Quantum Programming Languages, ENTCS*, 2006.

[Mau04] L. Mauborgne. Astrée: Verification of absence of run-time error. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 385–392. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[Min] A. Miné. The `Octagon` abstract domain library. `http://www.di.ens.fr/~mine/oct/`.

[Min01] A. Miné. A new numerical abstract domain based on difference-bound matrices. In 0. Danvy and A. Filinski, editors, *Proceedings of the Second Symposium PADO '2001, Programs as Data Objects*, Århus, Denmark, 21–23 May 2001, Lecture Notes in Computer Science 2053, pages 155–172. Springer, Berlin, Germany, 2001.

[Min04a] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proceedings of the Thirteenth European Symposium on Programming Languages and Systems, ESOP '2004, Barcelona, Spain*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, Berlin, Germany, March 27 – April 4, 2004.

[Min04b] A. Miné. *Weakly Relational Numerical Abstract Domains*. Thèse de doctorat en informatique, École polytechnique, Palaiseau, France, 6 December 2004.

[Min05] A. Miné. Weakly relational numerical abstract domains: Theory and application, invited paper. In *First International Workshop on Numerical & Symbolic Abstract Domains, NSAD '05*, Maison Des Polytechniciens, Paris, France, 21 January 2005.

[Min06a] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '2006*, pages 54–63. ACM Press, New York, New York, United States, June 2006.

[Min06b] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[Min06c] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E.A. Emerson and K.S. Namjoshi, editors, *Proceedings of the Seventh International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2006)*, pages 348–363, Charleston, South Carolina, United States, 8–10, January 2006. Lecture Notes in Computer Science 3855, Springer, Berlin, Germany.

[Mon05] D. Monniaux. The parallel implementation of the Astrée static analyzer. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems, APLAS '2005*, pages 86–96, Tsukuba, Japan, 3–5 November 2005. Lecture Notes in Computer Science 3780, Springer, Berlin, Germany.

[MR05] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proceedings of the Fourteenth European Symposium on Programming Languages and Systems, ESOP '2005, Edinburg, Scotland*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, Berlin, Germany, April 2—10, 2005.

[PCJD07] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. Semantics-based approach to malware detection. In *Conference Record of the Thirtyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Nice, France, 17–19 January 2007. ACM Press, New York, New York, United States.

[Per06] S. Perdrix. *Modèles formels du calcul quantique : ressources, machines abstraites et calcul par mesure*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire Leibniz, 2006.

[Riv05a] X. Rival. Abstract dependences for alarm diagnosis. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems, APLAS '2005*, pages 347–363, Tsukuba, Japan, 3–5 November 2005. Lecture Notes in Computer Science 3780, Springer, Berlin, Germany.

[Riv05b] X. Rival. Understanding the origin of alarms in Astrée. In C. Hankin and I. Siveroni, editors, *Proceedings of the Twelfth International Symposium on Static Analysis, SAS '05*, pages 303–319, London, United Kingdom, Lecture Notes in Computer Science 3672, 7–9 september 2005.

[RT04] F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proceedings of the Thirteenth European Symposium on Programming Languages and Systems, ESOP '04*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32, Barcelona, Spain, March 29 – April 2 2004. Springer, Berlin, Germany.

[RT06] F. Ranzato and F. Tapparo. Strong preservation of temporal fixpoint-based operators by abstract interpretation. In A.E. Emerson and K.S. Namjoshi, editors, *Proceedings of the Seventh International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2006)*, pages 332–347, Charleston, South Carolina, United States, 8–10 January 2006. Lecture Notes in Computer Science 3855 , Springer, Berlin, Germany.