« The ASTRÉE Static Analysis Tool »

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

ES_PASS Workshop — Berlin — 16–17 October 2007

Abstract

Software verification consists in proving that executions of the software in any admissible execution environment all satisfy a formal specification. In the case of the ASTRÉE static analyser (www.astree.ens.fr), the specification is implicit: no execution can lead to a "runtime error" (RTE) (such as bufer overrun, dangling pointer, division by zero, float overflow, modular integer arithmetic overflow, ...). The ASTRÉE static analyser is designed by abstract interpretation of the semantics of a subset of the C programming language (without dynamic memory allocation, recursive function calls, no system and library calls as found in most embedded software). Abstract interpretation is a theory of abstraction, that is to say of safe approximation allowing for automatic formal proofs by 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 the specification is mathematically valid. However, some execution 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. This spurious execution is then said to lead to a "false alarm". 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). In practice, knowing only the over-approximation computed by the static analyser, it is difficult to distinguish false alarms from actual ones. So the only effective solution is to reach zero false alarm which provides a full verification. To do so, ASTRÉE is specialized both to precise program properties (i.e. RTEs) and a precise family of C programs (i.e. real-time synchronous control/command C applications preferably automatically generated from a synchronous language). The ASTRÉE static analyser uses generalist abstractions (like intervals, octagons, decision trees, symbolic execution, etc) and specific abstractions for the application domain (to cope with filters, integrators, slow divergences due to rounding errors, etc). These domain-specific abstractions allowed for the verification of the absence of RTEs in large avionic software, a world première.

Résumé

La vérification d'un logiciel consiste à démontrer que toutes les exécutions d'un programme satisfont une spécification. Dans le cas de l'analyseur statique ASTRÉE www.astree.ens.fr), la spécification est implicite : aucune exécution ne peut conduire à une "erreur à l'exécution" (débordement de tableau, pointeur indéfini, division par zéro, débordement arithmétique, etc.). L'interprétation abstraite est une théorie de l'abstraction, c'est-à-dire des approximations sûres permettant de faire la preuve automatiquement en considérant des sur-ensembles des comportements possibles du programme. Contrairement aux méthodes de recherche d'erreurs comme le model-checking borné ou le test, aucun cas possible n'est omis et la preuve d'erreurs à l'exécution est donc mathématiquement valide. Certaines exécutions dans la sur-approximation peuvent conduire à une erreur sans pour autant correspondre à une exécution réelle (encore dite concrète). On parle dans ce cas d'une "fausse alarme". Toute la difficulté du problème indécidable de la vérification est de choisir des approximations sûres sans aucune fausse alarme (les erreurs conduisent à de vraies alarmes ne peuvent être éliminées qu'en les corrigeant). Dans le cas d'ASTRÉE, les programmes écrits en C sont des logiciels synchrones de contrôle commande temps réel. ASTRÉE contient des abstractions généralistes (intervalles, octogones, etc) et des abstractions spécifiques au domaine d'application (avec filtres, intégrateurs, divergences lentes à cause d'erreurs d'arrondis, etc). Cette adaptation au domaine d'application a permis de vérifier formellement l'absence d'erreurs à l'exécution dans des logiciels avioniques critiques de grande taille, une première mondiale.

Contents

1. Motivation

A Strong Need for Software Better Quality

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

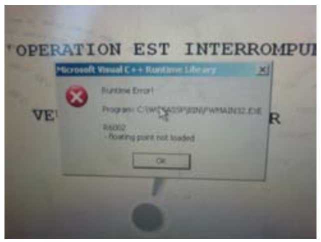



– The present state of the art in software engineering does not offer sufficient quality garantees

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):



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

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:
    130 000 LOCS (V1),
    250 000 LOCS (V2),
  - A380: 1.000.000 LOCS

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)

$\Rightarrow$ 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
– 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

Remedies to False Alarms in Astrée

– Astrée is specialized to specific program properties [2]
– Astrée is specialized to real-time synchronous control/command programs
– Astrée offers possibilities of refinement [3]

[2] proof of absence of runtime errors
[3] 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

References

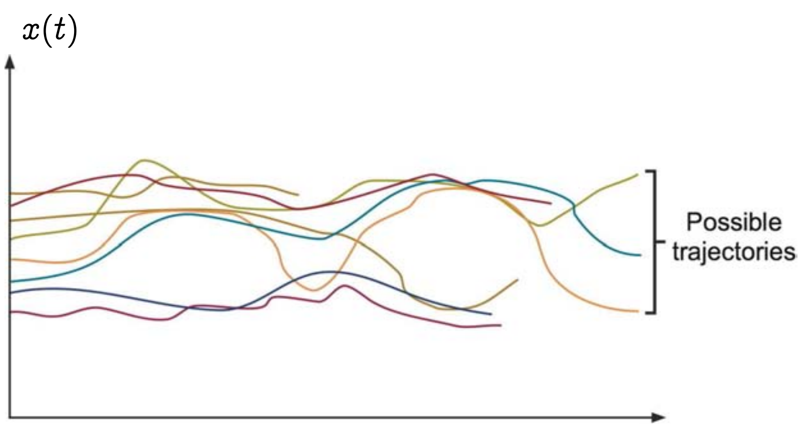
[POPL '77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM POPL.*

[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.

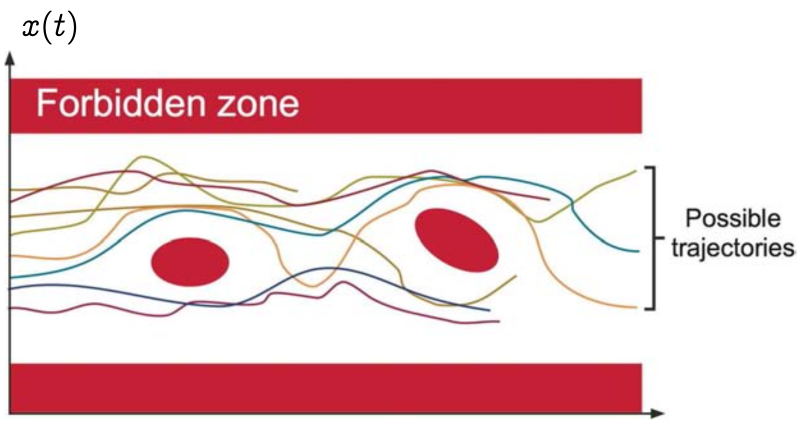
[POPL '79] P. Cousot & R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *ACM POPL.*

Principle of Abstraction

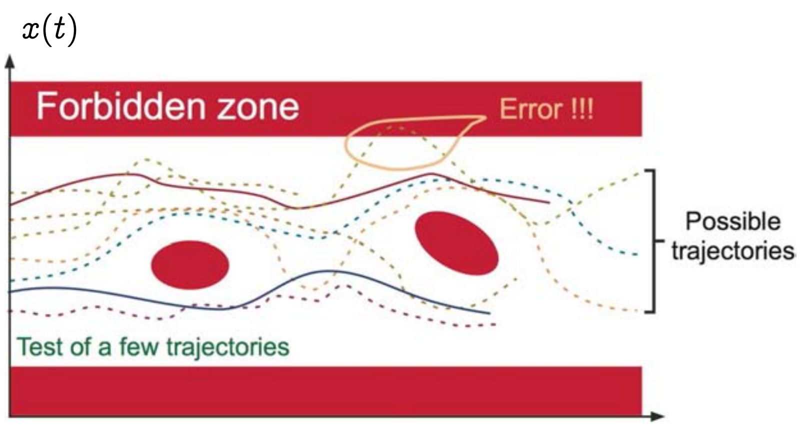
Operational semantics



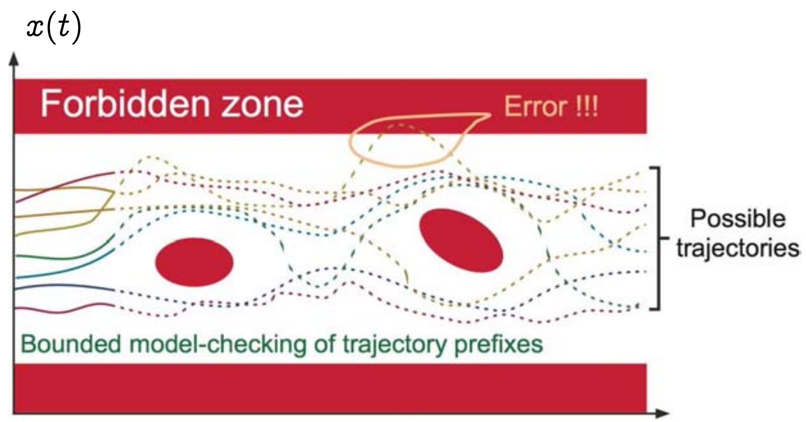
Safety property



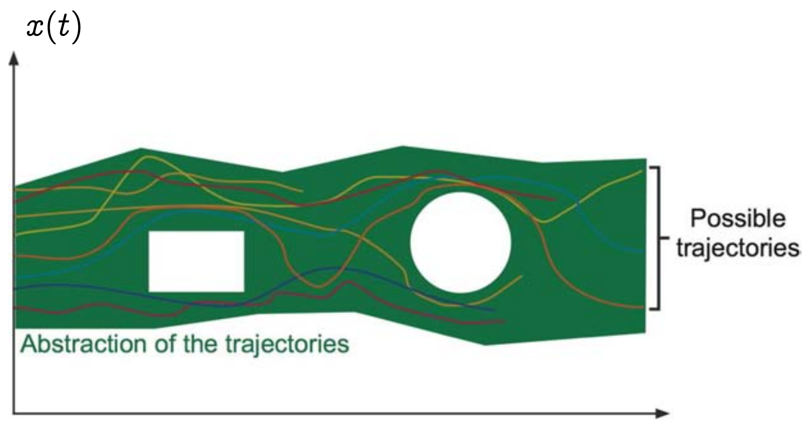
Test/Debugging is Unsafe



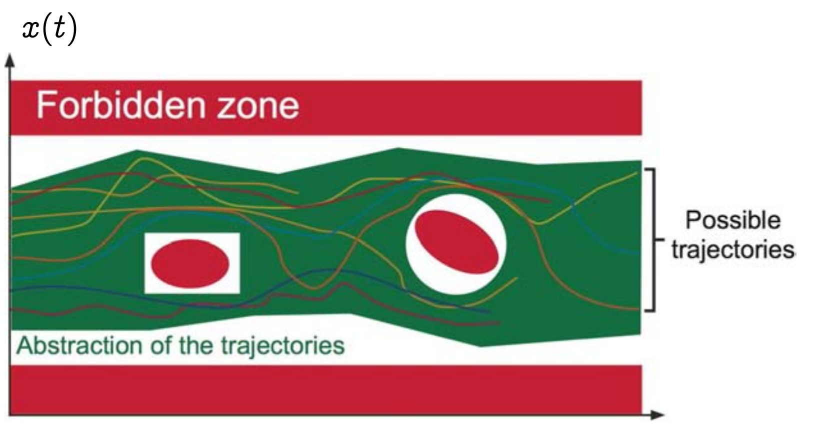
Bounded Model Checking is Unsafe



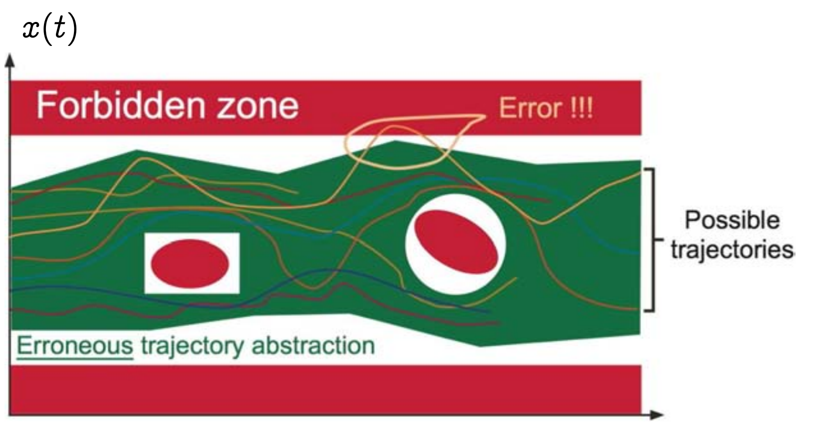
Over-Approximation



Abstract Interpretation is Sound



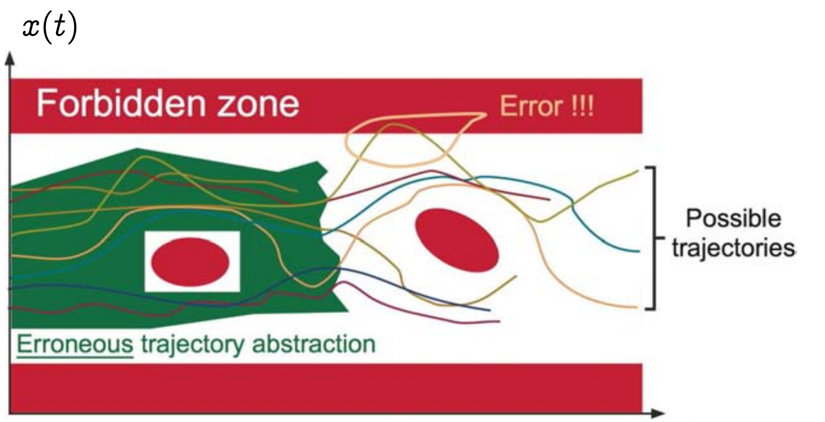
Soundness Requirement: Erroneous Abstraction [4]



4 This situation is always excluded in static analysis by abstract interpretation.

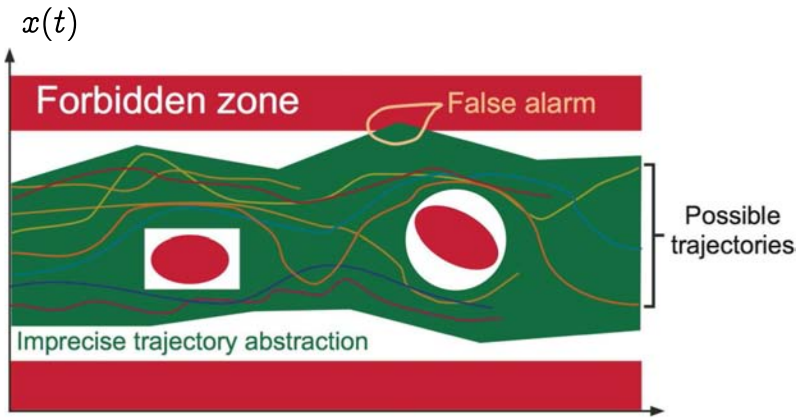
Soundness and Incompleteness

Soundness Requirement: Erroneous Abstraction [5]



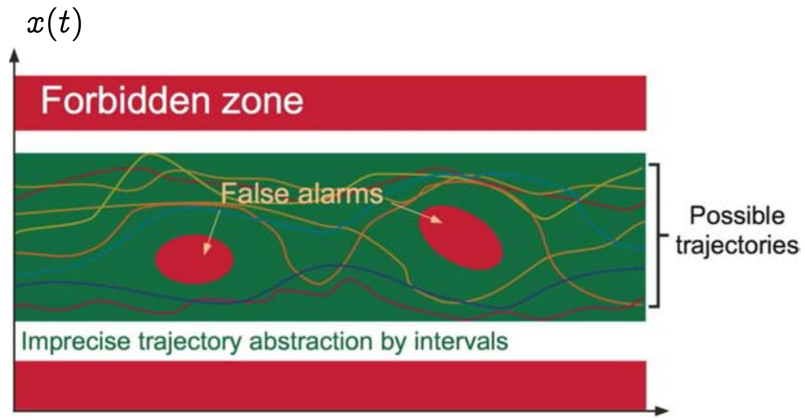
5 This situation is always excluded in static analysis by abstract interpretation.

Imprecision ⇒ False Alarms

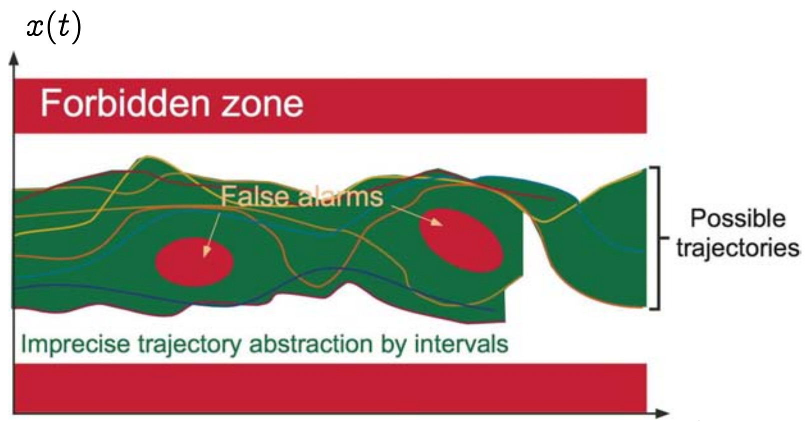


Design by Refinement

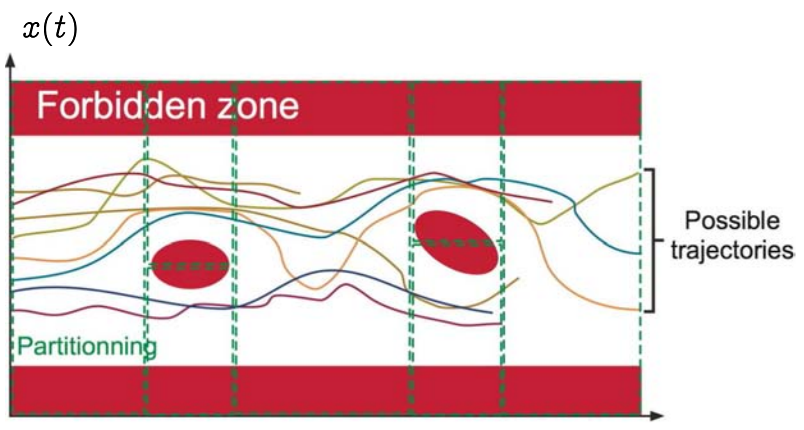
Global Interval Abstraction → False Alarms



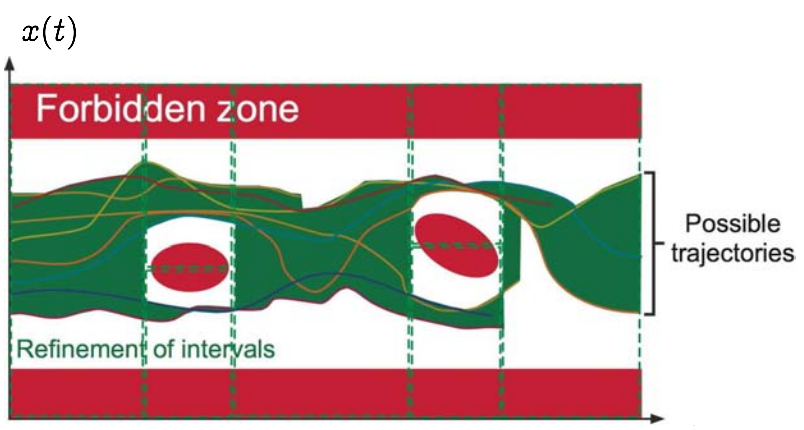
Local Interval Abstraction → False Alarms



Refinement by Partitionning



Intervals with Partitionning



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

Control Partitioning for Case Analysis

– Code Sample:

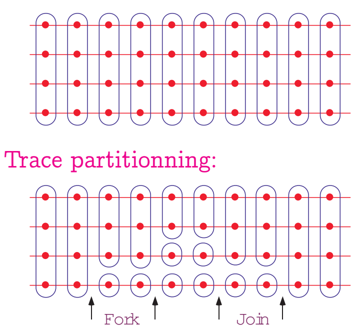
```
/* trace_partitionning.c */
void main() {
  float t[5] = {-10.0, -10.0, 0.0, 10.0, 10.0};
  float c[4] = {0.0, 2.0, 2.0, 0.0};
  float d[4] = {-20.0, -20.0, 0.0, 20.0};
  float x, r;
  int i = 0;

  ... found invariant −100 ≤ x ≤ 100 ...

  while ((i < 3) && (x >= t[i+1])) {
    i = i + 1;
  }
  r = (x - t[i]) * c[i] + d[i];
}
```

Control point partitionning:

Trace partitionning:



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

3. The ASTRÉE static analyzer

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

Project Members


Bruno BLANCHET[6]


Patrick COUSOT


Radhia COUSOT


Jérôme FERET


Laurent MAUBORGNE


Antoine MINÉ


David MONNIAUX[7]


Xavier RIVAL

[6] Nov. 2001 — Nov. 2003.
[7] 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:
  - <u>with</u>
    · 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)
  - union NEW [Min06a]
  - pointer arithmetics & casts NEW [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[8])

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

Different Classes of Run-time Errors

1. Errors terminating the execution [9]. 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 [10]. ASTRÉE warns and continues with worst-case assumptions.
3. Errors not terminating the execution with unpredictable outcome [11]. 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.

[9] floating-point exceptions e.g. (invalid operations, overflows, etc.) when traps are activated
[10] e.g. overflows over signed integers resulting in some signed integer.
[11] e.g. memory corruptionss.

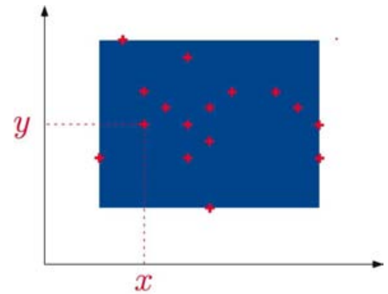
Specification Proved by Astrée

Examples of General Purpose Abstractions Used 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).

Intervals [12]

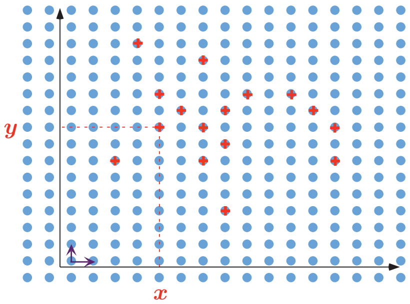


$$\begin{cases} x \in [19,\ 77] \\ y \in [20,\ 07] \end{cases}$$

Non-relational

[12] P. Cousot & R. Cousot. *Static determination of dynamic properties of programs.* Proc. 2^{nd} Int. Symp. on Programming, Dunod, 1976.

Simple congruences



$$\begin{cases} x = 19 \bmod 77 \\ y = 20 \bmod 99 \end{cases}$$

Non-relational

[13] Ph. Granger. *Static Analysis of Arithmetical Congruences*. Int. J. Comput. Math. 30, 1989, pp. 165–190.

Intervals & Simple Congruences

```
% cat -n congruence.c
/* congruence.c */
int main()
{ int X;
  X = 0;
  while (X <= 128)
    { X = X + 4; };
  __ASTREE_log_vars((X));
}
% astree congruence.c -no-relational -exec-fn main |& egrep "(WARN)|(X in)"
direct = <integers (intv+cong+bitfield+set): X in {132} /\ Top >
%
```


Symbolic abstract domain [Min04a, Min04b]

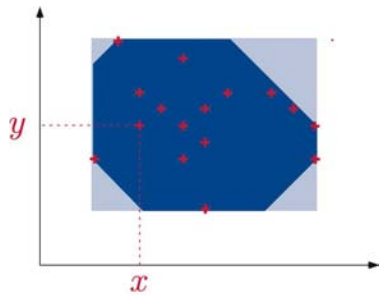
- Interval analysis: if $x \in [a, b]$ and $y \in [c, d]$ then $x - y \in [a - d, b - c]$ so if $x \in [0, 100]$ then $x - x \in [-100, 100]$!!!
- The symbolic abstract domain propagates the symbolic values of variables and performs simplifications;
- Must maintain the maximal possible rounding error for float computations (overestimated with intervals);

```
% cat -n x-x.c
void main () { int X, Y;
        __ASTREE_known_fact(((0 <= X) && (X <= 100)));
        Y = (X - X);
        __ASTREE_log_vars((Y));
}
```

```
astree -exec-fn main -no-relational x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:
<interval: Y in [-100, 100]>
```

```
astree -exec-fn main x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:
<interval: Y in {0}> <symbolic: Y = (X -i X)>
```


Octagons



$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 77 \\ 1 \leq y \leq 9 \\ x - y \leq 99 \end{cases}$$

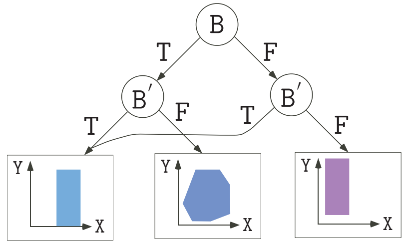
Weakly relational

[14] A. Miné. *A New Numerical Abstract Domain Based on Difference-Bound Matrices*. PADO'2001. LNCS 2053, pp. 155–172. Springer 2001. See the *The Octagon Abstract Domain Library* on http://www.di.ens.fr/~mine/oct/

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

(Automatic) Parameterization

– All abstract domains of ASTRÉE are parameterized, e.g.
  - variable packing for octagones and decision trees,
  - partition/merge program points,
  - loop unrollings,
  - thresholds in widenings, ...;
– End-users can either parameterize by hand (analyzer options, directives in the code), or
– choose the automatic parameterization (default options, directives for pattern-matched predefined program schemata).

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 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]
```


Float Overflow

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 destroyed after 40 seconds of flight because of a software overflow[13]

[13] 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
```

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

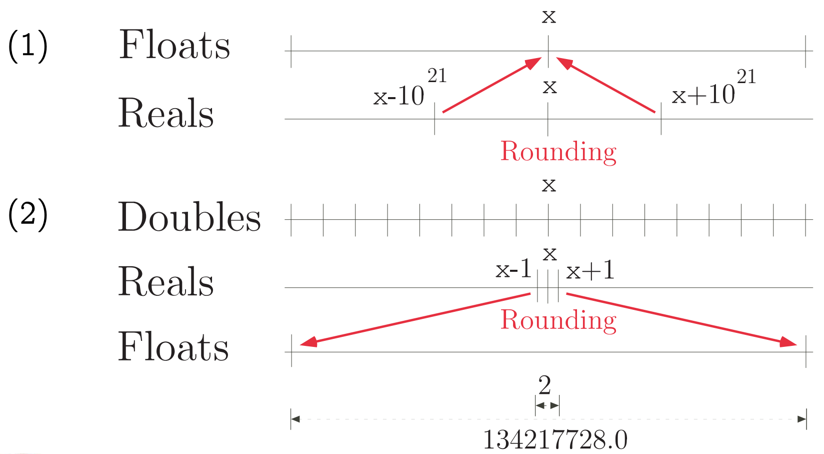
Example of rounding error

```
/* 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[14]

```
% cat -n 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;
__ASTREE_log_vars((r));
}
% 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] >
```

[14] 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
#include <stdio.h>
int main () {
 int i; double x; x = 0.0;
 for (i=1; i<=1000000000; i++) {
  x = x + 1.0/10.0;
 }
printf("x = %f\n", x);
}
% gcc rounding-c.c
% ./a.out
x = 99999998.745418
%
```

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

Static analysis with ASTRÉE

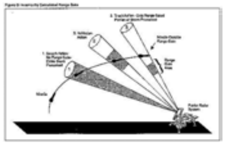
```
% cat -n rounding.c
int main () {
 double x; x = 0.0;
 while (1) {
  x = x + 1.0/10.0;
  __ASTREE_log_vars((x));
  __ASTREE_wait_for_clock(());
 }
}
% cat rounding.config
 __ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem rounding.config -unroll 0 rounding.c\
 |& egrep "(x in)|(\|x\|)|(WARN)" | 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”

Scaling

Static Analysis of Scaling with ASTRÉE

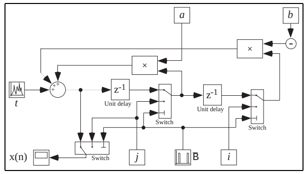
```
% cat -n scale.c
 1 int main () {
 2  float x; x = 0.70000001;
 3  while (1) {
 4   x = x / 3.0;
 5   x = x * 3.0;
 6   __ASTREE_log_vars((x));
 7   __ASTREE_wait_for_clock(());
 8  }
 9 }
```

```
% 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] >
%
```


Filtering

2^d Order Digital Filter:



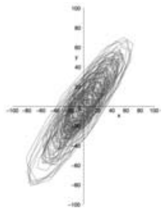
Ellipsoid Abstract Domain for Filters

– Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$

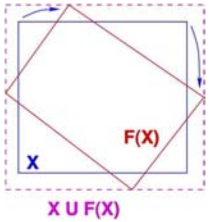
– The concrete computation is bounded, which must be proved in the abstract.

– There is no stable interval or octagon.

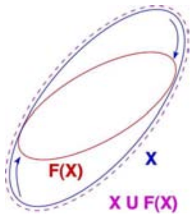
– The simplest stable surface is an ellipsoid.



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

Arithmetic-Geometric Progressions (Example 1)

```
% 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.
```

Arithmetic-Geometric Progressions: Example 2

```
% 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 progressions [15] [Fer05]

– 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.

References

[1] J. Feret. The arithmetic-geometric progression abstract domain. In *VMCAI'05*, Paris, LNCS 3385, pp. 42–58, Springer, 2005.

[17] here in TR

4. The industrial use of ASTRÉE

References

[2] D. Delmas and J. Souyris. ASTRÉE: *from Research to Industry*. Proc. 14^{th} 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 $\times$ 2

– A380: $\times$ 3/7

Digital Fly-by-Wire Avionics[16]

[18] 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[17], 132,000 lines, 75,000 LOCs after preprocessing
- Comparative results (commercial software):

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

  0 alarms,

  40mn on 2.8 GHz PC, 300 Megabytes

  $\longrightarrow$ A world première in Nov. 2003!

[19] "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) $\times$ 75,000 LOCs.

(Airbus A380 Primary Flight Control Software)

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

  34h,

  8 Gigabyte

  $\longrightarrow$ 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 [18];
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract ⟶ add a new abstract domain to the reduced product (e.g. filters).

[20] 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);
- More applications are forthcoming (ES_PASSS project);
- Industrialization is simultaneously under consideration.

THE END, THANK YOU

6. Bibliography

[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*, LNCS 2566, pages 85–108. Springer, 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 *Proc. ACM SIGPLAN '2003 Conf. PLDI*, pages 196–207, San Diego, CA, US, 7–14 June 2003. ACM Press.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In M. Sagiv, editor, *Proc. 14th ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 21–30. Springer, 2–10 Apr. 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, *11th ASIAN 06*, Tokyo, JP, 6–8 Dec. 2006. LNCS , Springer. 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, *Proc. 1st TASE '07*, pages 3–17, Shanghai, CN, 6–8 June 2007. IEEE Comp. Soc. Press.

[Cou07] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code, invited tutorial. In *Proc. 7th Int. Conf. EMSOFT '2007*, LNCS. Springer, 2007. To appear.

[DS07] D. Delmas and J. Souyris. ASTRÉE: from research to industry. In G. Filé and H. Riis-Nielson, editors, *Proc. 14th Int. Symp. SAS '07*, Kongens Lyngby, DK, LNCS 4634, pages 437–451. Springer, 22–24 Aug. 2007.

[Fer04] J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 27 – Apr. 4, 2004.

[Fer05] J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Proc. 6th Int. Conf. VMCAI 2005*, pages 42–58, Paris, FR, 17–19 Jan. 2005. LNCS 3385, Springer.

[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, *Proc. 1st Int. Work. EMSOFT '2001*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.

[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 Acad. Pub., 2004.

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

[Min04a] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 27 – Apr. 4, 2004.

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

[Min05] A. Miné. Weakly relational numerical abstract domains: Theory and application, invited paper. In *1st Int. Work. on Numerical & Symbolic Abstract Domains, NSAD '05*, Maison Des Polytechniciens, Paris, FR, 21 Jan. 2005.

[Min06a] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. LCTES '2006*, pages 54–63. ACM Press, 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, *Proc. 7th Int. Conf. VMCAI 2006*, pages 348–363, Charleston, SC, US, 8–10, Jan. 2006. LNCS 3855, Springer.

[Mon05] D. Monniaux. The parallel implementation of the ASTRÉE static analyzer. In *Proc. 3rd APLAS '2005*, pages 86–96, Tsukuba, JP, 3–5 Nov. 2005. LNCS 3780, Springer.

[MR05] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proc. 14th ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 2—-10, 2005.

[Riv05a] X. Rival. Abstract dependences for alarm diagnosis. In *Proc. 3rd APLAS '2005*, pages 347–363, Tsukuba, JP, 3–5 Nov. 2005. LNCS 3780, Springer.

[Riv05b] X. Rival. Understanding the origin of alarms in ASTRÉE. In C. Hankin and I. Siveroni, editors, *Proc. 12th Int. Symp. SAS '05*, pages 303–319, London, UK, LNCS 3672, 7–9 Sep. 2005.