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 analyzer (www.astree.ens.fr), the specification is implicit: no execution can lead to a "runtime error" (RTE) (such as buffer overrun, dangling pointer, division by zero, float overflow, modular integer arithmetic overflow, ...). The *Astrée* static analyzer 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 analyzer, 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 analyzer uses generalist abstractions (like intervals, octagones, 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, octagones, decision trees, symbolic execution, 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**

- Motivation .................................................. 5
- Informal introduction to abstract interpretation ........ 13
- The *Astrée* static analyzer ........................ 33
- The industrial use of *Astrée* ......................... 73
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. 21\(^{st}\) 2006 at 8:30 am):

**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 guarantees

**Tool-Based Software Design Methods**
- New tool-based software design methods will have to emerge to face the unprecedented growth and complexity 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

\(^1\) cash machine, cash dispenser, automatic teller machine.
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, unrealistic 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
- **ASTRÉE** is specialized to real-time synchronous control/command programs
- **ASTRÉE** offers possibilities of refinement
2. Informal Introduction to 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


Principle of Abstraction

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.

Operational semantics
Safety property

Bounded Model Checking is Unsafe

Test/Debugging is Unsafe

Over-Approximation
Abstract Interpretation is Sound

Soundness Requirement: Erroneous Abstraction

Soundness Requirement: Erroneous Abstraction

Soundness and Incompleteness
Imprecision ⇒ False Alarms

Global Interval Abstraction ⇒ False Alarms

Local Interval Abstraction ⇒ False Alarms

Design by Refinement
Refinement by Partitioning

\[ x(t) \]

Forbidden zone

Refinement of intervals

Trace Partitioning

Principle:

- Semantic equivalence:

  \[
  \begin{align*}
  &\text{if (B) \{} \ C1 \} \text{ else } \{ C2 \}; \ C3 \\
  &\downarrow \\
  &\text{if (B) \{} \ C1; C3 \} \text{ else } \{ C2; C3 \};
  \end{align*}
  \]

- More precise in the abstract: concrete execution paths are \textit{merged later}.

Application:

\[
\begin{align*}
&\text{if (B)} \\
&\quad \{ \ X=0; \ Y=1; \} \\
&\text{else} \\
&\quad \{ \ X=1; \ Y=0; \} \\
&\quad R = 1 \div (X-Y);
\end{align*}
\]

Intervals with Partitioning

\[ x(t) \]

Forbidden zone

Trace Partitioning

Control Partitioning for Case Analysis

Code Sample:

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

Control point partitioning:

Trace partitioning:

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  
Patrick Cousot  
Radhia Cousot  
Jérôme Feret  
Laurent Mauborgne  
Antoine Miné  
David Monniaux  
Xavier Rival

Programs Analyzed by ASTRÉE and their Semantics

- 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)
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)

\[\text{such limitations are quite common for embedded safety-critical software.}\]

The Class of Considered Periodic Synchronous Programs

\[\text{declare volatile input, state and output variables;}\]
\[\text{initialize state and output variables;}\]
\[\text{loop forever}\]
\[\text{- read volatile input variables,}\]
\[\text{- compute output and state variables,}\]
\[\text{- write to output variables;}\]
\[\text{__ASTREE\_wait\_for\_clock ();}\]
\[\text{end loop}\]

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].

Different Classes of Run-time Errors

1. Errors terminating the execution. \(\text{ASTR\`EE}\) 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. \(\text{ASTR\`EE}\) warns and continues with worst-case assumptions.
3. Errors not terminating the execution with unpredictable outcome. \(\text{ASTR\`EE}\) warns and continues by taking into account only the executions that did not trigger the error.

\(\Rightarrow\) \(\text{ASTR\`EE}\) is sound with respect to C standard, unsound with respect to C implementation, unless no false alarm.
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

\[
\begin{align*}
\forall x & \in [19, 77] \\
\forall y & \in [20, 07]
\end{align*}
\]

Non-relational

---

Simple congruences

\[
\begin{aligned}
    x &= 19 \text{ mod } 77 \\
    y &= 20 \text{ mod } 99
\end{aligned}
\]

Non-relational

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 congruence.c
1 /* congruence.c */
2 int main()
3 { int X;
4   X = 0;
5   while (X <= 128)
6     { X = X + 4; };
7   __ASTREE_log_vars((X));
8 }
% astree congruence.c –no-relational
```

Intervals & Simple Congruences

```
% cat -n congruence.c
1 /* congruence.c */
2 int main()
3 { int X;
4   X = 0;
5   while (X <= 128)
6     { X = X + 4; };
7   __ASTREE_log_vars((X));
8 }
% astree congruence.c –no-relational
```

Octagons

```
1 \leq x \leq 9
x + y \leq 77
1 \leq y \leq 9
x - y \leq 99
```

Weakly relational

```
% cat -n congruence.c
1 /* congruence.c */
2 int main()
3 { int X;
4   X = 0;
5   while (X <= 128)
6     { X = X + 4; };
7   __ASTREE_log_vars((X));
8 }
% astree congruence.c –no-relational
```
Boolean Relations for Boolean Control

- Code Sample:

```c
/* 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 ASTREE 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 (int+cong+bitfield+set); y in [-2147483648, 2147483647] \ Top
x in (2147483647) \ (2147483647) >

ASTRÉE signals the overflow and goes on with an unknown 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 }
7 %. astree -exec-fn main
overflow.c | & grep "WARN"
overflow.c:3.4-23::[call#main@1::]: WARN: double arithmetic range
[1.79769e+308, inf] not
included in [-1.79769e+308, 1.79769e+308]
overflow.c:4.4-24::[call#main@1::]: WARN: double arithmetic range
[-inf, -1.79769e+308] not
included in [-1.79769e+308, 1.79769e+308]

%. gcc overflow.c
%. ./a.out
x = inf, y = -inf

**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 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 uncontrollable.

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

(x + a) − (x − a) ≠ 2a

/* double-error.c */
int main () {
    double x; float y, z, r;
    /* x = ldexp(1.,50)+ldexp(1.,26); */
    x = 1.125899973951487.0;
    y = x + 1;
    z = x - 1;
    r = y - z;
    printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
1.34217728.000000

ES_PASS Workshop, 16-17/10/2007 — 56 — © P. Cousot

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

(x + a) − (x − a) ≠ 2a

/* double-error.c */
int main () {
    double x; float y, z, r;
    /* x = ldexp(1.,50)+ldexp(1.,26); */
    x = 1.125899973951487.0;
    y = x + 1;
    z = x - 1;
    r = y - z;
    printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
1.34217728.000000

ES_PASS Workshop, 16-17/10/2007 — 58 — © P. Cousot

ES_PASS Workshop, 16-17/10/2007 — 57 — © P. Cousot
**Explanation of the huge rounding error**

(1) Floats
Reals

\[
x - 10^{21} \quad x \quad x + 10^{21}
\]

Rounding

(2) Doubles
Reals
Floats

\[
x - 1 \quad x \quad x + 1
\]

Rounding

---

**Example of accumulation of small rounding errors**

\[
\text{since } (0.1)_{10} = (0.0001100110011001100...)_2
\]

---

**Static analysis with ASTRÉE**

\[
\% \text{ cat -n double-error.c}
\]

1. #include <stdio.h>
2. int main () {
3.    double x; x = 0.0;
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. }

---

**Static analysis with ASTRÉE**

\[
\% \text{ cat -n rounding.c}
\]

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. }

---

**Static analysis with ASTRÉE**

\[
\% \text{ cat -n rounding.config}
\]

\[
\% \text{ astree --exec-fn main --config-sem rounding.config --unroll 0 rounding.c} \\& \text{ egrep "(x in)|(|x|)|(WARN)" | tail -2}
\]

---

**Static analysis with ASTRÉE**

\[
\% \text{ cat -n rounding.config}
\]

\[
\% \text{ astree --exec-fn main --config-sem rounding.config --unroll 0 rounding.c} \\& \text{ egrep "(x in)|(|x|)|(WARN)" | tail -2}
\]

---

**Example of accumulation of small rounding errors**

\[
\% \text{ cat -n rounding-c.c}
\]

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. }

\[
\% \text{ gcc rounding-c.c}
\]

\[
\% \text{ ./a.out}
\]

\[
\% \text{ astree --exec-fn main --print-float-digits 10 double-error.c} \\& \text{ grep "r in"} \text{ | tail -1}
\]

---

**Example of accumulation of small rounding errors**

\[
\% \text{ cat -n rounding.c}
\]

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. }

\[
\% \text{ cat rounding.config}
\]

\[
\% \text{ astree --exec-fn main --config-sem rounding.config --unroll 0 rounding.c} \\& \text{ egrep "(x in)|(|x|)|(WARN)" | tail -2}
\]

---

**Example of accumulation of small rounding errors**

\[
\% \text{ cat -n rounding.config}
\]

\[
\% \text{ astree --exec-fn main --config-sem rounding.config --unroll 0 rounding.c} \\& \text{ egrep "(x in)|(|x|)|(WARN)" | tail -2}
\]

---

**Example of accumulation of small rounding errors**

\[
\% \text{ cat -n rounding.config}
\]

\[
\% \text{ astree --exec-fn main --config-sem rounding.config --unroll 0 rounding.c} \\& \text{ egrep "(x in)|(|x|)|(WARN)" | tail -2}
\]
The Patriot missile failure

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

<i>(i) This Scud subsequently hit an Army barracks, killing 28 Americans.</i>

Static Analysis of Scaling with ASTRÉE

```bash
% cat 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

% 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] >
```

Scaling

Filtering
Ellipsoid Abstract Domain for Filters
- Computes \( X_n = \frac{aX_{n-1} + \beta X_{n-2} + Y_n}{L_n} \)
- 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.

Filter Example [Fer04]

```c
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; }
}
```

Arithmetic-Geometric Progressions (Example 1)

```c
% 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.
```

Time Dependence
Arithmetic-Geometric Progressions: Example 2

```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(); 
  }
}
```

```c
% 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
```

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

References


4. The industrial use of Astrée

Arithmetic-geometric progressions

- Abstract domain: \((\mathbb{R}^+)\)^5
- Concretization:
  \[ \gamma \in (\mathbb{R}^+) \times (\mathbb{R}^+) \times (\mathbb{R}^+) \times (\mathbb{R}^+) \times (\mathbb{R}^+) \]
  \[ \gamma(M, a, b, a', b') = \{ f \mid \forall k \in \mathbb{N} : |f(k)| \leq (\lambda x \cdot ax + b \circ (\lambda x \cdot a'x + b')^k)(M) \} \]
  i.e. any function bounded by the arithmetic-geometric progression.

References

Digital Fly-by-Wire Avionics

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, 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
  → A world première in Nov. 2003!

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 \((x + \text{clk} \in [a; b] \land x - \text{clk} \in [a; b])\)
  - 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
  → 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 \(^\text{18}\);
- **Inexpressivity** i.e. indispensable local inductive invariant are inexpressible in the abstract → add a **new abstract domain** to the reduced product (e.g. filters).

\(^\text{18}\) 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.

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)

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.
6. Bibliography


