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 buffer 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 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. Such spurious executions are 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 radical solution is to reach zero false alarm which provides a full verification. To do so, Astrée is specialised 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 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 RTEs in several large avionic software, a world première.
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 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

\(^{1}\) 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)
  ⇒ **High costs** (for specification, proof assistance, etc).

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

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

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

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

Operational semantics

\[ x(t) \]
Safety property

Test/Debugging is Unsafe

Bounded Model Checking is Unsafe

Over-Approximation
Abstract Interpretation is Sound

Soundness Requirement: Erroneous Abstraction

Soundness and Incompleteness

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

5 This situation is always excluded in static analysis by abstract interpretation.
Imprecision $\Rightarrow$ False Alarms

Global Interval Abstraction $\Rightarrow$ False Alarms

Local Interval Abstraction $\Rightarrow$ False Alarms

Design by Refinement
Refinement by Partitionning

Principle:
- Semantic equivalence:
  \[
  \text{if (B) \{ C1 \} else \{ C2 \}; C3}
  \]
  
  \[
  \text{if (B) \{ C1; C3 \} else \{ C2; C3 \};}
  \]
- More precise in the abstract: concrete execution paths are merged later.

Application:

\[
\text{if (B)} \\
\text{\{ X=0; Y=1; \}} \\
\text{else} \\
\text{\{ X=1; Y=0; \}}
\]

\[ R = \frac{1}{(X-Y)}; \]

cannot result in a division by zero

Trace Partitioning

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 \(-100 \leq x \leq 100\) ...

  while ((i < 3) && (x >= t[i+1])) {
    i = i + 1;
  }

  r = (x - t[i]) * c[i] + d[i];
  
  Control point 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/

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)

Project Members

Bruno Blanchet\(^6\)  Patrick Cousot  Radhia Cousot  Jérôme Feret
Laurent Auborgne\(^6\)  Antoine Miné\(^7\)  David Monniaux\(^7\)  Xavier Rival

\(^6\) Nov. 2001 — Nov. 2003
\(^7\) Nov. 2001 — Aug. 2007.
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

The Class of Considered Periodic Synchronous Programs

```c
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].

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 corruptions.
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).

Rencontres INRIA–Industrie, 11/10/2007 — 42 — © P. Cousot

Modular Arithmetic

Modular arithmetics is not very intuitive

In C:

```c
#include <stdio.h>

int main () {
    int x,y;
    x = -2147483647 / -1;
    y = ((-x) -1) / -1;
    printf("x = %i, y = %i\n",x,y);
}
```

% 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

Specifications Proved by ASTRÉE
Static Analysis with ASTREÉ

% 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} >

ASTREÉ 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 
% 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 destroyed after 40 seconds of flight because of a software overflow  

12 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.

Example of rounding error

```c
/* 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
```

```c
/* 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
```

Example of rounding error

```c
/* 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
```

```c
/* 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) ≠ 2a
### Example of accumulation of small rounding errors

% cat -n rounding-c.c
1 #include <stdio.h>
2 int main () {
3  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 double-error.c
1 int main () {
2  double x; float y, z, r;
3  /* x = ldexp(1.,50)+ldexp(1.,26); */
4  x = 1125899973951488.0;
5  y = x + 1;
6  z = x - 1;
7  r = y - z;
8  __ASTREE_log_vars((r));
9  __ASTREE_wait_for_clock();
10 }
% gcc double-error.c
% 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 [-134217728, 134217728] >

---

### Static analysis with ASTRÉE

% cat -n double-error.c
1 int main () {
2  double x; float y, z, r;
3  /* x = ldexp(1.,50)+ldexp(1.,26); */
4  x = 1125899973951488.0;
5  y = x + 1;
6  z = x - 1;
7  r = y - z;
8  __ASTREE_log_vars((r));
9  __ASTREE_wait_for_clock();
10 }
% gcc double-error.c
% 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 [134217728, 134217728] >

---

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

- “On February 25th, 1991, a Patriot missile ... failed to track and inter-
cept an incoming Scud (i).”
- The software failure was due to accumu-
lated rounding error (i)

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

(ii) “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 inac-
curacy in the time calculation caused the range gate to shift
so much that the system could not track the incoming Scud.”

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

% cat scale.config
__ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem scale.config -unroll 0 scale.c\n|& grep "x in" | tail -1

direct = <float-interval: x in [0.69999986887, 0.700000047684] >
Ellipsoid Abstract Domain for Filters

- Computes \( X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \end{cases} \)
- 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.

Filter Example \([\text{Fer04}]\)

```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)

```bash
% 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; }
        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 *I. <= 3600001.
```
Arithmetic-Geometric Progressions: Example 2

```c
typedef enum {FALSE=0, TRUE=1} BOOL;
void main()
{
    FIRST = TRUE;
    while (TRUE) {
        dev();
        FIRST = FALSE;
        __ASTREE_wait_for_clock();
    }
}

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;}
}
```

% 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;}
}

% 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 - 5.8774715411e-39 / 1.19209290217e-07 <= 23.0393526881

4. The industrial use of ASTRÉE

References


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
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 \(^{17}\);
- **Inexpressivity** i.e. indispensable local inductive invariant are inexpressible in the abstract → add a **new abstract domain** to the reduced product (e.g. filters).

\(^{17}\) 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\(^{\text{TM}}\), 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!

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)

**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


