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

Operational semantics

$\text{Safety property}$

Test/Debugging is Unsafe

Bounded Model Checking is Unsafe
Over-Approximation

Abstract Interpretation is Sound

Soundness and Incompleteness

Soundness Requirement: Erroneous Abstraction

1 This situation is always excluded in static analysis by abstract interpretation.
Soundness Requirement: Erroneous Abstraction

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

2. 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

Programs analyzed 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)
  - **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

```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].
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)

Different Classes of Run-time Errors

1. Errors terminating the execution. \(^5\) \(\text{ASTRÉE}\) 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. \(\text{ASTRÉE}\) warns and continues with worst-case assumptions.
3. Errors not terminating the execution with unpredictable outcome. \(\text{ASTRÉE}\) warns and continues by taking into account only the executions that did not trigger the error.

\(\text{ASTRÉE}\) 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).
Examples of General Purpose Abstractions Used by ASTRÉE

Simple congruences

\[ x = 19 \mod 77 \]
\[ y = 20 \mod 99 \]

Non-relational

Intervals

\[ \{ x \in [19, 77] \} \]
\[ \{ y \in [20, 07] \} \]

Non-relational

Intervals & Simple Congruences

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);

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

---

**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 analyser option) and the abstract domain at the leaves.

---

**Octagons**

1. \( 1 \leq x \leq 9 \)
2. \( x + y \leq 77 \)
3. \( 1 \leq y \leq 9 \)
4. \( x - y \leq 99 \)

Weakly relational

---

(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 arithmetic 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);
}
```

```bash
% cat -n modulo.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 }

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

Static Analysis with ASTRÉE

```bash
% cat -n modulo.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 }

% astree -exec-fn main -unroll 0 modulo.c
```

```bash
Astrée signals the overflow and goes on with an unknown 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
```

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.

10 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.

The Ariane 5.01 maiden flight

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

Rounding
Example of accumulation of small rounding errors

```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);
}
```

since \(0.1_{10} = (0.000110011001100\ldots)_{2}\)

The Patriot missile failure

- “On February 25th, 1991, a Patriot missile . . . failed to track and intercept an incoming Scud (1).”
- The software failure was due to accumulated rounding error (2)

(1) This Scud subsequently hit an Army barracks, killing 28 Americans.
(2) “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”

Static analysis with Astrée

```c
int main () {
    double x; x = 0.0;
    while (1) {
        x = x + 1.0/10.0;
        __ASTREE_log_vars((x));
        __ASTREE_wait_for_clock();
    }
}
```

Filtering
Ellipsoid Abstract Domain for Filters
- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ \frac{X}{L} \end{cases}$
- 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.

Time Dependence

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.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 *1. <= 3600001.
```
Arithmetic-Geometric Progressions: Example 2

```c
% 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
```

### 3. The industrial use of ASTRÉE

#### 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**: $\times 3/7$

#### Benchmarks (Airbus A340 Primary Flight Control Software)

- V1 $^{13}$, 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!

---

References

(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
  \[\rightarrow \text{A world grand première!}\]

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)

**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

**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.
5. Bibliography


