Abstract Interpretation and Application to the Static Analysis of Mission-Critical Embedded Computer Software

Patrick Cousot
Patrick.Cousot@ens.fr  www.di.ens.fr/~cousot

Final review of the ESA ITI project « Space Software Validation using Abstract Interpretation » (SSVAI)

Noordwijk, The Netherlands, Sep. 19th, 2008
Abstract

Static software analysis has known brilliant successes in the small, by proving complex program properties of programs of a few dozen or hundreds of lines, either by systematic exploration of the state space or by interactive deductive methods. To scale up is a definite problem. Very few static analyzers are able to scale up to millions of lines without sacrificing automation and/or soundness and/or precision. Unsound static analysis may be useful for bug finding but is less useless in safety critical applications where the absence of bugs, at least of some categories of common bugs, should be formally verified.

After recalling the basic principles of abstract interpretation including the notions of abstraction, approximation, soundness, completeness, false alarm, etc., we introduce the domain-specific static analyzer ASTRÃE (www.astree.ens.fr) for proving the absence of runtime errors in mission critical real time embedded synchronous software in the large.

The talk emphasizes soundness (no runtime error is ever omitted), parametrization (the ability to refine abstractions by options and analysis directives), extensibility (the easy incorporation of new abstractions to refine the approximation), precision (few or no false alarms for programs in the considered application domain) and scalability (the analyzer scales to millions of lines).

In conclusion, present-day software engineering methodology, which is based on the control of the design, coding and testing processes should evolve in the near future, to incorporate a systematic control of final software product thanks to domain-specific analyzers that scale up.
1. Classical Examples of Bugs
Classical examples of bugs in integer computations
The factorial program (fact.c)

```c
#include <stdio.h>

int fact (int n ) {
    int r, i;
    r = 1;
    for (i=2; i<=n; i++) {
        r = r*i;
    }
    return r;
}

int main() {
    int n;
    scanf("%d",&n);
    printf("%d!=%d\n",n,fact(n));
}
```

- `fact(n) = 2 \times 3 \times \cdots \times n`
- read \( n \) (typed on keyboard)
- write \( n! = fact(n) \)
Compilation of the factorial program (fact.c)

```c
#include <stdio.h>

int fact (int n ) {
    int r, i;
    r = 1;
    for (i=2; i<=n; i++) {
        r = r*i;
    }
    return r;
}

int main() { int n;
    scanf("%d",&n);
    printf("%d!=%d\n",n,fact(n));
}
```

% gcc fact.c -o fact.exec
%
Executions of the factorial program (fact.c)

```
#include <stdio.h>

int fact (int n ) {
    int r, i;
    r = 1;
    for (i=2; i<=n; i++) {
        r = r*i;
    }
    return r;
}

int main() { int n;
    scanf("%d",&n);
    printf("%d!=%d\n",n,fact(n));
}

% gcc fact.c -o fact.exec
% ./fact.exec
  3
 3!  = 6
% ./fact.exec
  4
 4!  = 24
% ./fact.exec
 100
100! = 0
% ./fact.exec
  20
20! = 2102132736
```
Bug hunt

- Computers use integer modular arithmetics on $n$ bits (where $n = 16, 32, 64, \text{ etc}$)
- Example of an integer representation on 4 bits (in complement to two):

- Only integers between -8 and 7 can be represented on 4 bits
- We get $7 + 2 = -7$
  
  \[ 7 + 9 = 0 \]
The bug is a failure of the programmer

In the computer, the function $\text{fact}(n)$ coincide with $n! = 2 \times 3 \times \ldots \times n$ on the integers only for $1 \leq n \leq 12$: 
Proof of absence of runtime error by static analysis

% cat -n fact_lim.c
1 int MAXINT = 2147483647;
2 int fact (int n) {
3     int r, i;
4     if (n < 1) || (n = MAXINT) {
5         r = 0;
6     } else {
7         r = 1;
8         for (i = 2; i<=n; i++) {
9             if (r <= (MAXINT / i)) {
10                r = r * i;
11            } else {
12                r = 0;
13            }
14         }
15     }
16     return r;
17 }
18

% astree –exec-fn main fact_lim.c |& grep WARN
%
→ No alarm!
Examples of classical bugs in floating point computations
Mathematical models and their implementation on computers

- **Mathematical models** of physical systems use **real numbers**

- **Computer modeling languages** (like **SCADE**) use **real numbers**

- **Real numbers** are hard to represent in a computer (\( \pi \) has an infinite number of decimals)

- **Computer programming languages** (like **C** or **OCAML**) use **floating point numbers**
Floats

- **Floating point numbers** are a finite subset of the *rationals*

- For example one can represent *32 floats on 6 bits*, the 16 positive normalized floats spread as follows on the line:

- When real computations do not spot on a float, one must *round* the result to a close float
Example of rounding error (1)

\[(x + a) - (x - a) \neq 2a\]

```c
#include <stdio.h>

int main() {
    double x, a; float y, z;
    x = 1125899973951488.0;
    a = 1.0;
    y = (x+a);
    z = (x-a);
    printf("%f\n", y-z);
}
```

% gcc arrondi1.c -o arrondi1.exec
% ./arrondi1.exec
134217728.000000
%
Example of rounding error (2)

\[(x + a) - (x - a) \neq 2a\]

#include <stdio.h>

int main() {
    double x, a; float y, z;
    x = 1125899973951487.0;
    a = 1.0;
    y = (x+a);
    z = (x-a);
    printf("%f\n", y-z);
}

% gcc arrondi2.c -o arrondi2.exec
% ./arrondi2.exec
0.000000
%
Bug hunt (1)

Doubles
Reals
Floats

$x$

$x-1 \parallel x+1$

Rounding

$\frac{2}{134217728.0}$
Bug hunt (2)

Doubles
Reals
Floats

\[ x \]

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

Rounding

\[ 0.0 \]
\[ 2 \]
Proof of absence of runtime error by static analysis

% cat -n arrondi3.c
1 int main() {
2 double x; float y, z, r;;
3 x = 1125899973951488.0;
4 y = x + 1;
5 z = x - 1;
6 r = y - z;
7 __ASTREE_log_vars((r));
8 }

% astree –exec-fn main –print-float-digits 10 arrondi3.c |
|& grep "r in "
direct = <float-interval: r in [-134217728, 134217728] >

(1) ASTRÉE considers the worst rounding case (towards $+\infty$, $-\infty$, 0 or to the nearest) whence the possibility to obtain -134217728.
The verification is done in the worst case.

Doubles

Reals

Floats

\[ x \]

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

Rounding

\[ \pm 134217728.0 \]
Examples of bugs due to rounding errors

- The **patriot missile bug** missing Scuds in 1991 because of a software clock incremented by $\frac{1}{10}$ of a seconde ($(0,1)_{10} = (0,0001100110011001100\ldots)_2$ in binary)
- The **Excel 2007 bug**: $77.1 \times 850$ gives 65,535 but displays as 100,000! $(^2)$

![Table of floating point numbers and their binary representations](image)

$(^2)$ Incorrect float rounding which leads to an alignment error in the conversion table while translating 64 bits IEEE 754 floats into a Unicode character string. The bug appears exactly for six numbers between 65534.99999999995 and 65535 and six between 65535.99999999995 and 65536.
Bugs in the everyday numerical world
Bugs are frequent in everyday life

- **Bugs** proliferate in banks, cars, telephones, washing machines, ...

- Example (**bug in an ATM machine** located at 19 Boulevard Sébastopol in Paris, on 21 November 2006 at 8:30):

- **Hypothesis (Gordon Moore’s law revisited):** the number of software bugs in the world doubles every 18 months???

:-(
2. Program verification
Principle of program verification

- Define a **semantics** of the language (that is the effect of executing programs of the language)

- Define a **specification** (example: absence of runtime errors such as division by zero, un arithmetic overflow, etc)

- Make a **formal proof** that the semantics satisfies the specification

- Use a computer to **automate the proof**
Semantics of programs
Operational semantics of program $P$
Example: execution trace of \texttt{fact(4)}

```c
int fact(int n) {
    int r = 1, i;
    for (i=2; i<=n; i++) {
        r = r*i;
    }
    return r;
}
```

\begin{itemize}
\item $n \leftarrow 4; r \leftarrow 1$
\item $i \leftarrow 2; r \leftarrow 1 \times 2 = 1$
\item $i \leftarrow 3; r \leftarrow 2 \times 3 = 6$
\item $i \leftarrow 4; r \leftarrow 6 \times 4 = 24$
\item $i \leftarrow 5$
\item return 24
\end{itemize}
Program specification
Specification of program $P$

Forbiden zone

Possible trajectories

$\mathcal{X}(t)$

Specification[$P$]
Example of specification

```c
int fact (int n) {
    int r, i;
    r = 1;
    for (i=2; i<=n; i++) {
        r = r*i; ← no overflow of r*i
    }
    return r;
}
```
Formal proofs
Formal proof of program $P$

$x(t)$

Semantics[$P$] $\subseteq$ Specification[$P$]
Undecidability and complexity

- The mathematical proof problem is undecidable\(^{(3)}\)

- Even assuming finite states, the complexity is much too high for combinatorial exploration to succeed

- Example: 1.000.000 lines \(\times\) 50.000 variables \(\times\) 64 bits \(\simeq\) 10\(^{27}\) states

- Exploring 10\(^{15}\) states per second, one would need 10\(^{12}\) s > 300 centuries (and a lot of memory)!

\(^{(3)}\) there are infinitely many programs for which a computer cannot solve them in finite time even with an infinite memory.
Testing is incomplete

\[ x(t) \]

Forbidden zone Error !!!

Test of a few trajectories

Possible trajectories
3. Abstract Interpretation [1]

Reference

Abstraction of program $P$

$\text{Abstraction(Semantics}$[P])$

$x(t)$

Abstraction of the trajectories

Possible trajectories
Proof by abstraction

$\text{Forbidden zone}$

$\text{Abstraction of the trajectories}$

$\text{Abstraction(Semantics}[P]\text{)} \subseteq \text{Specificatlon}[P]$
Soundness of abstract interpretation
Abstract interpretation is sound

\[ x(t) \]

Forbidden zone

Abstraction of the trajectories

Semantics[\(P\)] ⊆ Abstraction(Semantics[\(P\)])
Example of unsound abstraction

Unsoundness is always excluded by abstract interpretation theory.
Unsound abstractions are inconclusive (false negatives) \((4)\)

\[ x(t) \]

(4) Unsoundness is always excluded by abstract interpretation theory.
Incompleteness of abstract interpretation
Alarm

$\mathbf{x(t)}$

Forbidden zone

Alarm !!!

Error or false alarm ?

Possible trajectories

$\mathbf{t}$
An alarm can originate from an error

\[ x(t) \]

Forbidden zone

Alarm !!!

Possible trajectories

Error
An alarm can originate from an over-approximation

\[ x(t) \]

Forbidden zone

Alarm !!!

False alarm

Possible trajectories
4. Applications of Abstract Interpretation
The Theory of Abstract Interpretation

- A theory of **sound approximation of mathematical structures**, in particular those involved in the behavior of computer systems
- Systematic derivation of **sound methods and algorithms for approximating undecidable or highly complex problems** in various areas of computer science
- **Main practical application** is on the **safety and security of complex hardware and software computer systems**
- **Abstraction**: extracting information from a system description that is relevant to proving a property
Applications of Abstract Interpretation

- **Static Program Analysis** (or Semantics-Checking) [CC77], [CH78], [CC79] including Dataflow Analysis; [CC79], [CC00], Set-based Analysis [CC95], Predicate Abstraction [Cou03], ...

- **Grammar Analysis and Parsing** [CC03];

- **Hierarchies of Semantics and Proof Methods** [CC92b], [Cou02];

- **Typing & Type Inference** [Cou97];

- **(Abstract) Model Checking** [CC00];

- **Program Transformation** (including compile-time program optimization, partial evaluation, etc) [CC02];
Applications of Abstract Interpretation (Cont’d)

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

All these techniques involve sound approximations that can be formalized by abstract interpretation.
5. Application of Abstract Interpretation to Static Analysis
Semantics

(Infinite) set of traces (finite or infinite)
Abstraction to a set of states (invariant)

Set of points \(\{(x_i, y_i) : i \in \Delta\}\), Floyd/Hoare/Naur invariance proof method [Cou02]
Abstraction by signs

Signs $x \geq 0, y \geq 0$  [CC79]
Abstraction by intervals

Intervals $a \leq x \leq b$, $c \leq y \leq d$  \[CC77\]
Abstraction by octagons

Octagons \( x - y \leq a, \ x + y \leq b \) \[\text{[Min06]}\]
Abstraction by polyedra

Polyedra $a \cdot x + b \cdot y \leq c$  [CH78]
Abstraction by ellipsoids

Ellipsoids $(x - a)^2 + (y - b)^2 \leq c$  [Fer05b]
Abstraction by exponentials

Exponentials $a^x \leq y$ [Fer05a]
6. Invariant Computation by Fixpoint Approximation [CC77]
\{ y \geq 0 \} \leftarrow \text{hypothesis} \\
x = y \\
\{ I(x, y) \} \leftarrow \text{loop invariant} \\
\text{while } (x > 0) \{ \\
\hspace{1em} x = x - 1; \\
\} \\
\text{Floyd-Naur-Hoare verification conditions:} \\
(y \geq 0 \land x = y) \implies I(x, y) \quad \text{initialisation} \\
(I(x, y) \land x > 0 \land x' = x - 1) \implies I(x', y) \quad \text{iteration} \\
\text{Equivalent fixpoint equation:} \\
I(x, y) = x \geq 0 \land (x = y \lor I(x + 1, y)) \quad \text{(i.e. } I = F(I)^{(5)}) \\
\hfill (5) \text{ We look for the most precise invariant } I, \text{ implying all others, that is } \text{lfp} \implies F.
Accelerated Iterates $I = \lim_{n \to \infty} F^n(false)$

$I^0(x, y) = \text{false}$

$I^1(x, y) = x \geq 0 \land (x = y \lor I^0(x + 1, y))$
  $= 0 \leq x = y$

$I^2(x, y) = x \geq 0 \land (x = y \lor I^1(x + 1, y))$
  $= 0 \leq x \leq y \leq x + 1$

$I^3(x, y) = x \geq 0 \land (x = y \lor I^2(x + 1, y))$
  $= 0 \leq x \leq y \leq x + 2$

$I^4(x, y) = I^2(x, y) \lor I^3(x, y)$ ← widening
  $= 0 \leq x \leq y$

$I^5(x, y) = x \geq 0 \land (x = y \lor I^4(x + 1, y))$
  $= I^4(x, y)$ fixed point!

The invariants are computer representable with octagons!
7. Scaling up
The difficulty of scaling up

- The abstraction must be coarse enough to be effectively computable with reasonable resources.

- The abstraction must be precise enough to avoid false alarms.

- Abstractions to infinite domains with widenings are more expressive than abstractions to finite domains (when considering the analysis of a programming language) [CC92a].

- Abstractions are ultimately incomplete (even intrinsically for some semantics and specifications [CC00]).
Abstraction/refinement by tuning the cost/precision ratio in Astrée

- Approximate reduced product of a choice of coarsenable/refinable abstractions
- Tune their precision/cost ratio by
  - Globally by parametrization
  - Locally by (automatic) analysis directives
so that the overall abstraction is not uniform.
Example of abstract domain choice in ASTRÉE

/* Launching the forward abstract interpreter */
/* Domains: Guard domain, and Boolean packs (based on Absolute value equality relations, and Symbolic constant propagation (max_depth=20), and Linearization, and Integer intervals, and congruences, and bitfields, and finite integer sets, and Float intervals), and Octagons, and High_passband_domain(10), and Second_order_filter_domain (with real roots)(10), and Second_order_filter_domain (with complex roots)(10), and Arithmetico-geometric series, and new clock, and Dependencies (static), and Equality relations, and Modulo relations, and Symbolic constant propagation (max_depth=20), and Linearization, and Integer intervals, and congruences, and bitfields, and finite integer sets, and Float intervals. */
Example of abstract domain functor in Astrée: decision trees

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 leaves.
Reduction [CC79, CCF+08]
Example: reduction of intervals [CC76] by simple congruences [Gra89]

% 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 –exec-fn main |& egrep "(WARN)|(X in)"
direct = <integers (intv+cong+bitfield+set): X in {132} >

Intervals : \(X \in [129, 132]\) + congruences : \(X = 0 \mod 4 \Rightarrow X \in \{132\}\).
Parameterized abstractions

- Parameterize the cost / precision ratio of abstractions in the static analyzer

- Examples:
  
  - array smashing: \texttt{--smash-threshold} \(n\) (400 by default)
    \(\rightarrow\) smash elements of arrays of size \(> n\), otherwise individualize array elements (each handled as a simple variable).
  
  - packing in octogons: (to determine which groups of variables are related by octagons and where)
    \(\bullet\) \texttt{--fewer-oct}: no packs at the function level,
    \(\bullet\) \texttt{--max-array-size-in-octagons} \(n\): unsmashed array elements of size \(> n\) don’t go to octagons packs
Parameterized widenings

- Parameterize the rate and level of precision of widenings in the static analyzer

- Examples:
  - **delayed widenings**: --forced-union-iterations-at-beginning $n$ (2 by default)
  
  - **thresholds for widening** (e.g. for integers):
    0; 1; 2; 3; 4; 5; 32767; 32768; 65535; 65536; 2147483647; 2147483648; 4294967295.
Analysis directives

– Enable a **local refinement of an abstract domain**

– Example:

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

    while (b) {
        x = x - 1;
        b = (x > 0);
    }
}

% astree -exec-fn main repeat1.c |& egrep "WARN"
repeat1.c:5.8-13::[call#main@2:loop@4>=4::]: WARN: signed int arithmetic range [-2147483649, 2147483646] not included in [-2147483648, 2147483647]
%```
Example of directive (Cont’d)

```c
% cat repeat2.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
    int x = 100; BOOL b = TRUE;
    __ASTREE_boolean_pack((b,x));
    while (b) {
        x = x - 1;
        b = (x > 0);
    }
}

% astree -exec-fn main repeat2.c |& egrep "WARN"
%

The insertion of this directive could be automated in ASTRÉE (if the considered family of programs has “repeat” loops).
```
Automatic analysis directives

- The directives can be inserted automatically by static analysis

- Example:

```c
% cat p.c
int clip(int x, int max, int min) {
    if (max >= min) {
        if (x <= max) {
            max = x;
        }
        if (x < min) {
            max = min;
        }
    }
    return max;
}
void main() {
    int m = 0; int M = 512; int x, y;
    y = clip(x, M, m);
    __ASTREE_assert(((m<=y) && (y<=M)));
}
% astree -exec-fn main p.c -dump-partition
... int (clip)(int x, int max, int min) {
    if ((max >= min))
    {
        __ASTREE_partition_control((0))
        if ((x <= max))
        {
            max = x;
        }
        if ((x < min))
        {
            max = min;
        }
        __ASTREE_partition_merge_last();
    }
    return max;
}
% astree -exec-fn main p.c |& grep WARN
...
Adding new abstract domains

- The weakest invariant to prove the specification may not be expressible with the current refined abstractions ⇒ false alarms cannot be solved

- No solution, but adding a new abstract domain:
  - representation of the abstract properties
  - abstract property transformers for language primitives
  - widening
  - reduction with other abstractions

- Examples: ellipsoids for filters [Fer05b], exponentials for accumulation of small rounding errors [Fer05a], quaternions, ...
Abstraction by ellipsoid for filters

Ellipsoids \((x - a)^2 + (y - b)^2 \leq c\) \([\text{Fer05b}]\)
Example of analysis by ASTRÉE

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; }
    }
Abstraction by exponentials for accumulation of small rounding errors

Exponentials $a^x \leq y$
Example of analysis by ASTRÉE

% 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
8. Industrial Application of Abstract Interpretation
Examples of static analyzers in industrial use

- For C critical synchronous embedded control/command programs (for example for Electric Flight Control Software)

- aiT \([FHL^{+01}]\) is a static analyzer to determine the Worst Case Execution Time (to guarantee synchronization in due time)

- ASTRÉE \([BCC^{+03}]\) is a static analyzer to verify the absence of runtime errors
Industrial results obtained with ASTRÉE

Automatic proofs of absence of runtime errors in Electric Flight Control Software:

– Software 1 : 132.000 lines of C, 40mn on a PC 2.8 GHz, 300 megabytes (nov. 2003)

– Software 2 : 1.000.000 lines of C, 34h, 8 gigabytes (nov. 2005)

no false alarm

World premières!
9. Conclusion on Future Prospects of Abstract Interpretation
Evolution of Software Engineering

- **State of the Art in Software Engineering**: Manual validation by control of the software design process
- **Desirable Evolution of Software Engineering**: Automatic verification of the final product
Challenges in Abstract Interpretation

Short term: automatic help on the diagnosis of the origin of alarms

Midterm: Parallelism

Long term: Liveness for infinite state systems
THE END

Thank you for your attention
10. Bibliography
Short bibliography


