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 useful in safety critical applications where the absence of bugs at all is of critical importance, and soundness and completeness are mandatory.

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 ASTRAÉ (www.astreame.fr) for proving the absence of runtime errors in safety 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 approximations), precision (few or no false alarms for programs in the considered application domain) and scalability (the analyzer scales to millions of lines).

In conclusion, prescience software engineering methodology, which is based on the control of the design, coding and testing process, should evolve to 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

Compilation 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));
}
```

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));
    // read n (typed on keyboard)
    // write n! = fact(n)
}
```

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));
    // read n (typed on keyboard)
    // write n! = fact(n)
}
```

```
% gcc fact.c -o fact.exec
% ./fact.exec
3
3! = 6
4
4! = 24
100
100! = 0
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):

  ![Integer representation on 4 bits](image)

  - Only integers between -8 and 7 can be represented on 4 bits
  - We get $7 + 2 = -7$
  - $7 + 9 = 0$

**And in OCAML, the result is different!**

```ocaml
let rec fact n = if (n = 1) then 1 else n * fact(n-1);;
```

| C | OCAML |
|---|---|---|---|
| fact(0) | \(-5^2735356\) | fact(22) | \(-52735356\) |
| fact(1) | 1 | fact(23) | \(82453536\) |
| fact(2) | -1 | fact(24) | \(-75956300\) |
| fact(3) | -1 | fact(25) | \(-75956300\) |
| fact(4) | -1 | fact(26) | \(-23456140\) |
| fact(5) | -1 | fact(27) | \(-23456140\) |
| fact(6) | -1 | fact(28) | \(-45478348\) |
| fact(7) | -1 | fact(29) | \(-45478348\) |
| fact(8) | -1 | fact(30) | \(-148478348\) |
| fact(9) | -1 | fact(31) | \(-148478348\) |
| fact(10) | -1 | fact(32) | \(-148478348\) |
| fact(11) | -1 | fact(33) | \(-148478348\) |
| fact(12) | -1 | fact(34) | \(-148478348\) |

**Why? What is the result of fact(-1)?**

**The bug is a failure of the programmer**

In the computer, the function $\text{fact}(n)$ coincide with $n! = 2 \times 3 \times \cdots \times n$ on the integers only for $1 \leq n \leq 12$:

```ocaml
% cat -n fact.c
1 int MAXINT = 234567890;
2 int fact (int n) {
3     int r, i;
4     if (n < 1) || (n = MAXINT) { r = 0; }
5     else {
6         r = 1;
7         for (i = 2; i<=n; i++) {
8             r = r * i;
9             if (r =< (MAXINT / i)) {
10                 r = r / i;
11             } else {
12                 r = 0;
13             }
14         }
15     }
16     return r;
17 }
```

**Proof of absence of runtime error by static analysis**

```
19 int main0 ()
20 { int n, f;
21     f = fact(n);
22 }
```

% arccp -o=aln min fact.c & grep WARN

% -> No alarm!
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 (π has an infinite number of decimals)

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

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);
}
```

Example of rounding error (2)

- 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:

```
0 1 2 3 4 5 6 7
```

- When real computations do not spot on a float, one must round the result to a close float
Example of rounding error (2)

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

```c
#include <stdio.h>
int main() {
    double x, a, z; float y, z;
    x = 1125899973951487.0;
    a = 1.0;
    y = (x+a);
    z = (x-a);
    printf("%f\n", y-z);
}
```

Bug hunt (2)

Doubles

Reals

Floats

Rounding

\[x\]

\[\frac{x}{2}\]

\[0.0\]

\[\frac{1}{2}\]

\[1\]

Proof of absence of runtime error by static analysis

```c
% cat -n arondi3.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 arondi3.c \n|& grep "r in "
direct = <float-interval: r in [-134217728, 134217728]>
```

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

Doubles | | | | | | | | | | |
Reals    | | | | | | | | | | |
Floats   | | | | | | | | | | |

\[ x \]

\[ x-1 \]

Rounding

\[ \frac{2}{4} \]

\[ =1.34217728.0 \]

Bugs in the everyday numerical world

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 second \((0,1)_{10} = (0,0001100110011001100_{2})\) in binary.
- The **Excel 2007 bug**: \(77.1 \times 850\) gives 65,535 but displays as 100,000!(2)

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):

   ![ATM Bug](image)

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

---

(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 the numbers between 0x3f40000000000000 and 0x3f43f40000000000 and also between 0x3f47000000000000 and 0x3f43f40000000000.
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

Operational semantics of program $P$
Example: execution trace of fact(4)

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

```
  n ← 4; r ← 1;
  i ← 2; r ← 1 × 2 = 1;
  i ← 3; r ← 2 × 3 = 6;
  i ← 4, r ← 6 × 4 = 24;
  i ← 5;
  return 24;
```

Specification of program P

![Specification diagram](image)

Example of specification

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

```
  ← no overflow of i++
  ← no overflow of r*i
```

Program specification
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: \(1,000,000 \text{ lines} \times 50,000 \text{ variables} \times 64 \text{ bits} \approx 10^{27}\) 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.
3. Abstract Interpretation [1]
Abstract interpretation is sound

Semantics[P] ⊆ Abstraction(Semantics[P])

Unsound abstractions are inconclusive (false negatives)

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

Example of unsound abstraction

Incompleteness of abstract interpretation
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 (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

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];

5. Application of Abstract Interpretation to Static Analysis
**Semantics**

(Infinite) set of traces (finite or infinite)

**Abstraction by signs**

Signs $x \geq 0, y \geq 0$  [CC79]

**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 intervals**

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

Octagons $x - y \leq a$, $x + y \leq b$ [Min06]

Abstraction by ellipsoids

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

Abstraction by polyhedra

Polyhedra $a \cdot x + b \cdot y \leq c$ [CH78]

Abstraction by exponentials

Exponentials $a^x \leq y$ [Fer05a]
6. Invariant Computation by Fixpoint Approximation [CC77]

\[ \text{Accelerated Iterates } I = \lim_{n \to \infty} F^n(\text{false}) \]
\[ I^0(x, y) = \text{false} \]
\[ I^1(x, y) = x \geq 0 \land (x = y \lor I^0(x + 1, y)) \]
\[ = 0 \leq x = y \]
\[ I^2(x, y) = x \geq 0 \land (x = y \lor I^1(x + 1, y)) \]
\[ = 0 \leq x \leq x + 1 \]
\[ I^3(x, y) = x \geq 0 \land (x = y \lor I^2(x + 1, y)) \]
\[ = 0 \leq x \leq x + 2 \]
\[ I^4(x, y) = I^3(x, y) \lor I^3(x, y) \leftarrow \text{widening} \]
\[ = 0 \leq x \leq y \]
\[ I^5(x, y) = x \geq 0 \land (x = y \lor I^4(x + 1, y)) \]
\[ = I^4(x, y) \text{ fixed point!} \]

The invariants are computer representable with octagons!

Fixpoint equation

\{ y \geq 0 \} \leftarrow \text{hypothesis} \\
x = y \\
\{ I(x, y) \} \leftarrow \text{loop invariant} \\
\text{while} (x > 0) \{ \\
\quad x = x - 1; \\
\}

Floyd-Naur-Hoare verification conditions:

\( y \geq 0 \land x = y \implies I(x, y) \) \hspace{1cm} \text{initialisation}
\( I(x, y) \land x > 0 \land x' = x - 1 \implies I(x', y) \) \hspace{1cm} \text{iteration}

Equivalent fixpoint equation:

\[ I(x, y) = x \geq 0 \land (x = y \lor I(x + 1, y)) \]
\hspace{1cm} \text{(i.e. } I = F(I))^{(0)}

\hspace{1cm} ^{(0)} \text{We look for the most precise invariant } I, \text{ implying all others, that is } \lim_{n \to \infty} F^n.

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**

```c
/* 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 Arithmetic-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]

```c
% 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 }
```

Intervals: $X \in [129, 132]$ + congruences: $X \equiv 0 \mod 4 \implies X \in \{132\}$.

---

**Parameterized widenings**

- Parameterize the rate and level of precision of widenings in the static analyzer
- Examples:
  - **delayed widenings**: -forced-union-iterations-at-beginning $\mathbb{N}$ (2 by default)
  - **thresholds for widening** (e.g. for integers):
    ```c
    let widening_sequence = [
      of_int 0, of_int 1, of_int 2, of_int 3, of_int 4, of_int 5,
      of_int 32767, of_int 32768, of_int 65535, of_int 65536,
      of_string "2147483647", of_string "2147483648",
      of_string "4294967295" ]
    ```

---

**Parameterized abstractions**

- Parameterize the cost / precision ratio of abstractions in the static analyzer
- Examples:
  - **array smashing**: -smash-threshold $n$ (400 by default)
    → smash elements of arrays of size > $n$, otherwise individuate array elements (each handled as a simple variable).
  - **packing in octogs**: (to determine which groups of variables are related by octogs and where)
    - --fewer-oct: no packs at the function level,
    - --max-array-size-in-octogs $n$: unsmashed array elements of size > $n$ don’t go to octogs packs

---

**Analysis directives**

- Require 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);
  }
}
```

---

% cat repeat1.c
```c
typedef enum {FALSE,TRUE} BOOL;
```
Example of directive (Cont’d)

% 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 Astree (if the considered family of programs has “repeat” loops).

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, ...

Automatic analysis directives

- The directives can be inserted automatically by static analysis
- Example:

  % cat c.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 n = 0; int M = 512; int x, y;
    y = clip(x, M, n);
    _ASTREE_assert({(m<n) & (y=m)});
  }
  % astree -exec-fn main c.c -dump-partition

  % astree -exec-fn main p.c -dump-partition
  ...
  int clip(int x, int max, int min) {
    if (max >= min) {
      if (x <= max) {
        max = x;
      }
      if (x < min) {
        max = min;
      }
    }
    return max;
  }
  ...
  %

Abstraction by ellipsoid for filters

\[ \pi(t) \]

Ellipsoids \((x - a)^2 + (y - b)^2 \leq c\) [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] * 0.6)) + (S[0] * 1.5)) - (S[1] * 0.6)); }  
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
    /* S[0], S[1] in [-1927.02698354, 1927.02698354] */
}

void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (!) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE; }  
}

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

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

- aIT [PHL01] is a static analyzer to determine the **Worst Case Execution Time** (to guarantee synchronization in due time)

- ASTRÉE [BCC03] is a static analyzer to verify the absence of runtime errors

---

9. Present and Future Work

---

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 !

---

Foundational Work

- Formalization of the descriptions of the behavior of discrete/hybrid complex systems (6) and mecanisation of the reasonings on such systems in terms of abstract interpretation

- Abstraction of numerical (7), symbolic (8) and control-flow (9) properties.

---

[6] image analysis [Fov], biological systems [DPP05, BPP06, Rok], quantum calculus [JF06], etc

[7] for example efficient and correct implementation of polyhedra with floats

[8] for example
  - low level memory models
  - complex dynamic data structures
  - cryptographic protocols

[9] for example, quasi-synchronism, concurrency, ...
Technological Transfer

- Widening of the application domain of ASTRÉE (space, aircraft engines, automobile, rail, telecommunications)
- Certification of ASTRÉE (for the aeronautic industry)
- Industrialisation of ASTRÉE

Challenges

Short term: Help of the diagnostic of origin of alarms
Midterm: Parallelism
Long term: Liveness for infinite systems

Conclusion

- **Vision**: to understand the numerical world, different levels of abstraction must be considered
- **Theory**: abstract interpretation ensures the coherence between abstractions and offers effective approximation techniques to cope with infinite systems
- **Applications**: the choice of effective abstraction which are coarse enough to be computable and precise enough to be avoid false alarms is central to master undecidability and complexity in model and program verification
The future

- Software engineering: Manual validation by control of the software design process will be complemented by the verification of the final product.

- Complex systems: Abstract interpretation applies equally well to the analysis of systems with discrete/hybrid evolution (image analysis [Ser94], biological systems [DFPK07, DFFK08, Fer07], quantum computation [JP06], etc).

11. Bibliography

Short bibliography


THE END

Thank you for your attention.


Answers to questions

- The integers are encoded on 32 bits in C and on 31 bits in OCaml (one bit is used for garbage collection).

- The call of `fact(-1)` calls `fact(-2)` which calls `fact(-3)`, etc. For each call, it is necessary to stack the parameter and return address, which ends by a stack overflow:

  ```ocaml
  % ocaml
  Objective Caml version 3.10.0
  # let rec fact n = if (n = 1) then 1 else n * fact(n-1);;
  val fact : int -> int = <fun>
  # fact(-1);;
  Stack overflow during evaluation (looping recursion?).
  ```