Abstract

Since almost any large complex software has bugs which are not found by test methods, researchers have developed program correctness proof methods which have been successful in the small. This consists in defining a semantics formally describing the executions of a program and then in proving a theorem stating that these executions have a given property (for example that an expected result is provided in a finite time). Fundamental mathematical undecidability results show that these proofs cannot be done automatically by computers.

Confronted with this fundamental difficulty, abstract interpretation proceeds by correct approximation of the semantics. If the approximation is sound, no potential error can ever be overlooked, a basic requirement of formal verification methods. If the approximation is coarse enough, it is computable. If it is precise enough, it yields a correctness proof. The goal is therefore to find cheap approximations (so as to scale up in the large) which are precise enough (to avoid false alarms where a property does hold but this cannot be proved because of an approximation which is too imprecise).

We will introduce a few elements of abstract interpretation and explain how to formalize the abstraction of semantic properties so as to obtain computable approximations leading to effective algorithms for the static analysis of the possible behaviors of programs.

Finally, we will describe an example of application of the theory to the proof of absence of runtime errors on synchronous control/command and underly the difficulties (such as floating point computations). This approach was applied with success to the verification of the electric flight control of the commercial planes.
Software is hidden everywhere

Software is massively present in all mission-critical and safety-critical industrial infrastructures

Accident analysis (metro)

- Paris métro line 12 accident\(^1\): the driver was going too fast

- Roma metro line A accident\(^2\): the driver went on at a red signal

\(^1\) On August 30th, 2000, at the Notre-Dame-de-Lorette métro station in Paris, a car flipped over on its side and slid to a stop just a few feet from a train stopped on the opposite platform (24 injured).

\(^2\) On October 17th, 2006, a speeding subway train rammed into another train halted at the Vittorio Emanuele station in central Rome (1 dead, 60 injured). The driver might have misunderstood the control centre authorising the train to proceed to the "next station" (Manzoni, closed to the public) while the driver would have understood it to mean the "next working station" (Vittorio Emanuele, after Manzoni), La Repubblica, Oct. 20th, 2006.

Accident analysis (avionics)

Primary cause of major commercial jet accidents world-wide as determined by the investigation authorities between 1995 and 2004\(^3\) [1]

![Diagram showing accident causes]

<table>
<thead>
<tr>
<th>Cause</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traffic control</td>
<td>56%</td>
</tr>
<tr>
<td>Maintenance</td>
<td>6%</td>
</tr>
<tr>
<td>Other</td>
<td>4%</td>
</tr>
<tr>
<td>Airplane</td>
<td>17%</td>
</tr>
<tr>
<td>Weather</td>
<td>13%</td>
</tr>
<tr>
<td>Flight crew</td>
<td>5%</td>
</tr>
</tbody>
</table>

Reference


\(^3\) Includes only accidents with known causes.

Software replaces human operators

- Computer control is the cheapest and safest solution to avoid such accidents

- New high-speed métro line 14 (Météor): fully automated, no operators

- Modern commercial airplanes: massive automation of control/commands, piloting, communications, collision avoidance, etc
Why is software erroneous?

As computer hardware capacity grows...

ENIAC
5,000 flops\textsuperscript{4}

NEC Earth Simulator
$35 \times 10^{12}$ flops\textsuperscript{5}

\textsuperscript{4} Floating point operations per second

\textsuperscript{5} $10^{12}$ = Thousand Billion

Software size grows...

(1) Software is huge

Text editor
1,700,000 lines of C\textsuperscript{6}

Operating system
35,000,000 lines of C\textsuperscript{7}

\textsuperscript{6} 3 months for full-time reading of the code

\textsuperscript{7} 5 years for full-time reading of the code
... and so does the number of bugs

Text editor
- 1,700,000 lines of C
- 1,700 bugs (estimation)

Operating system
- 35,000,000 lines of C
- 30,000 known bugs

---

Computers are finite

- Scientists reason on continuous, infinite mathematical structures (e.g. \( \mathbb{R} \))
- Computers can only handle discrete, finite structures

Overflows

- Numbers are encoded onto a limited number of bits (binary digits)
- Some operations may overflow (e.g. integers: 32 bits \( \times \) 32 bits = 64 bits)
- Using different number sizes (32, 64, ... bits) can also be the source of overflows
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

Modular arithmetic...
- Today's computers avoid integer overflows thanks to **modular arithmetic**
- Example: integer 2’s complement encoding on 8 bits

---

8 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.
... can be contrary to common sense

```plaintext
# 1073741823 + 1;;
- : int = -1073741824
# -1073741824 - 1;;
- : int = 1073741823
# -1073741824 ÷ -1;;
- : int = -1073741824
```

<table>
<thead>
<tr>
<th>Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Floats: mapping many to few

- Reals are mapped to floats (floating-point arithmetic) \( \pm d_0 . d_1 d_2 \ldots d_{p-1}\beta^e \)

- For example on 6 bits (with \( p = 3, \beta = 2, e_{\min} = -1, e_{\max} = 2 \)), there are 32 normalized floating-point numbers. The 16 positive numbers are

\( d_0 \neq 0, \quad p \) is the number of significative digits, \( \beta \) is the basis (2), and \( e \) is the exponent \( (e_{\min} \leq e \leq e_{\max}) \)

Rounding

- Computations returning reals that are not floats, must be rounded
- Most mathematical identities on \( \mathbb{R} \) are no longer valid with floats
- Rounding errors may either compensate or accumulate in long computations
- Computations converging in the reals may diverge with floats (and ultimately overflow)
Example of rounding error (1)

```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

(x + a) - (x - a) ≠ 2a
```

/* double-error.c */
int main () {
  double x, y, z, r;
  x = 125899973951488.0;
  y = x + 1;
  z = x - 1;
  r = y - z;
  printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
0.000000

Explanation of the huge rounding error

(1) Floats
Reals
Rounding

\( x-10^{21} \)
\( x \)
\( x+10^{21} \)

(2) Doubles
Reals
Floats

Example of accumulation of small rounding errors

```ml
% ocaml
Objective Caml version 3.08.1

# let x = ref 0.0;;
val x : float ref = {contents = 0.}

# for i = 1 to 1000000000 do
  x := !x +. 1.0/.10.0
done; x;;
- : float ref = {contents = 99999997.454178184}
since (0.1)_{10} = (0.0001100110011001100...)_{2}
```
The Patriot missile failure

– The software failure was due to a cumulated rounding error.

10 This Scud subsequently hit an Army barracks, killing 28 Americans.
11 – “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”

Warranty

Excerpt from an GPL open software licence:

NO WARRANTY. … BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

You get nothing for free!

Absence of Warranty

Excerpt from Microsoft software licence:

DISCLAIMER OF WARRANTIES. … MICROSOFT AND ITS SUPPLIERS PROVIDE THE SOFTWARE, AND SUPPORT SERVICES (IF ANY) AS IS AND WITH ALL FAULTS, AND MICROSOFT AND ITS SUPPLIERS HEREBY DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY (IF ANY) IMPLIED WARRANTIES, DUTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF RELIABILITY OR AVAILABILITY, OF ACCURACY OR COMPLETENESS OF RESPONSES, OF RESULTS, OF WORKMANLIKE EFFORT, OF LACK OF VIRUSES, AND OF LACK OF NEGLIGENCE, ALL WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE OR OTHERWISE ARISING OUT OF THE USE OF THE SOFTWARE …

You get nothing for your money either!
Mathematics and computers can help

- Software behavior can be mathematically formalized → semantics
- Computers can perform semantics-based program analyses to realize verification → static analysis
  - but computers are finite so there are intrinsic limitations → undecidability, complexity
  - which can only be handled by semantics approximations → abstract interpretation

Traditional software validation methods

- The law cannot enforce more than “best practice”
- Manual software validation methods (code reviews, simulations, tests, etc.) do not scale up
- The capacity of programmers/computer scientists remains essentially the same
- The size of software teams cannot grow significantly without severe efficiency losses

Abstract interpretation

There are two fundamental concepts in computer science (and in science in general):

- **Abstraction**: to reason on complex systems
- **Approximation**: to make effective undecidable computations

These concepts are formalized by **Abstract interpretation**.

References


Abstract interpretation
(1) Very informal introduction

Safety property

Operational semantics

Test/debugging is unsafe
Abstract interpretation is safe

\[ x(t) \]

Forbidden zone

Possible trajectories

Abstraction of the trajectories

\[ t \]

Imprecision ⇒ false alarms

\[ x(t) \]

Forbidden zone False alarm

Possible trajectories

Imprecise trajectory abstraction

\[ t \]

Soundness requirement: erroneous abstraction

\[ x(t) \]

Forbidden zone Error !!!

Possible trajectories

Erroneous trajectory abstraction

\[ t \]

Global interval abstraction → false alarms

\[ x(t) \]

Forbidden zone False alarms

Possible trajectories

Imprecise trajectory abstraction by intervals

\[ t \]

\[ 12 \] This situation is always excluded in static analysis by abstract interpretation.
Local interval abstraction → false alarms

Forbidden zone

Imprecise trajectory abstraction by intervals

Refinement by partitionning

Forbidden zone

Partitionning

Intervals with partitioning

Imprecise trajectory abstraction by intervals

Refinement of intervals

Abstract interpretation
(2) A few elements of AI
(2.1) Program semantics

Description of a computation step

- Transition system \((\Sigma, \tau)\), states \(\Sigma = \{\bullet, \ldots, \bullet\ldots\}\), transitions \(\tau = \{\bullet \rightarrow \bullet, \ldots, \bullet \rightarrow \bullet\ldots\}\)

- Example
  - States : \(\langle p, v \rangle\), \(p\) is a program point, \(v\) assigns values to variables
  - Transitions \(\langle p, v \rangle \rightarrow \langle p', v' \rangle\) for assignment:
    \[
    \begin{align*}
    p': & \ x = x + 1; \\
    p': & \ v'(x) = v(x) + 1 \text{ si } v(x) < \text{maxint} \\
    p': & \ v'(y) = v(y) \quad \text{si } y \neq x
    \end{align*}
    \]
  - Blocking state (\(\bullet\)) if \(v(x) \geq \text{maxint}\).

Least Fixpoint Trace Semantics

\[
\begin{align*}
\text{Traces} & = \{ \bullet \mid \bullet \text{ is a final state} \} \\
& \cup \{ \bullet \rightarrow \bullet \rightarrow \ldots \rightarrow \bullet \mid \bullet \rightarrow \bullet \text{ is a transition step} \} \\
& \cup \{ \bullet \rightarrow \bullet \rightarrow \ldots \rightarrow \bullet \mid \bullet \rightarrow \bullet \text{ is a transition step} \}
\end{align*}
\]

- In general, the equation has multiple solutions;
- Choose the least one for the computational ordering:
  "more finite traces \& less infinite traces".

States \(\Sigma = \{\bullet, \ldots, \bullet\ldots\}\), transitions \(\tau = \{\bullet \rightarrow \bullet, \ldots, \bullet \rightarrow \bullet\ldots\}\)
Iterative Fixpoint Calculation of the Trace Semantics

<table>
<thead>
<tr>
<th>Iterates</th>
<th>Finite traces</th>
<th>Infinite traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P^0$</td>
<td>$\emptyset$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td>$P^1$</td>
<td>${\cdot}$</td>
<td>${\tau, \ldots}$</td>
</tr>
<tr>
<td>$P^2$</td>
<td>${\cdot, \tau}$</td>
<td>${\tau, \tau, \ldots}$</td>
</tr>
<tr>
<td>$P^3$</td>
<td>${\cdot, \tau, \tau}$</td>
<td>${\tau, \tau, \tau, \ldots}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>${\ldots}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td>$P^n$</td>
<td>${\ldots}$</td>
<td>${\ldots}$</td>
</tr>
<tr>
<td>$P^\omega$</td>
<td>${\ldots}$</td>
<td>${\ldots}$</td>
</tr>
</tbody>
</table>

(2.2) Program Properties

Program Properties & Static Analysis
- A program property $P \in \wp(D)$ is a set of possible semantics for that program (hence a subset of the semantic domain $D$)
- A property $P \in \wp(D)$ is stronger (or more precise) than a property $Q \in \wp(D)$ iff $P \subseteq Q$ (i.e. $P$ implies $Q$, $P \Rightarrow Q$)
- The strongest program property \(^{13}\) is $\{S\}$
- A static analysis effectively approximates the strongest property of programs

\(^{13}\) also called the collecting semantics
Example of program property

- Correct implementations: print 0, print 1, [print 1\|loop], ...
- Incorrect implementations: [print 0|print 1]

Abstraction

- Replace actual concrete properties $\mathcal{P} \in \wp(D)$ by an approximate abstract properties $\alpha(\mathcal{P})$
- Example:
  - $D = \wp(\Sigma^+ \cup \Sigma^\omega)$
  - $\mathcal{P} \in \wp(D)$
  - $\alpha(\mathcal{P}) \overset{\text{def}}{=} \wp(\bigcup \mathcal{P})$

Commonly Required Properties of the Abstraction

- [In this talk,] we consider overapproximations: $\mathcal{P} \subseteq \alpha(\mathcal{P})$
  - If the abstract properties $\alpha(\mathcal{P})$ is true then the concrete properties $\mathcal{P}$ is also true
  - If the abstract properties $\alpha(\mathcal{P})$ is false then the concrete properties $\mathcal{P}$ may be true or false!
- All information is lost at once: $\alpha(\alpha(\mathcal{P})) = \alpha(\mathcal{P})$
- The abstraction of more precise properties is more precise: $\text{si } \mathcal{P} \subseteq \mathcal{Q} \text{ alors } \alpha(\mathcal{P}) \subseteq \alpha(\mathcal{Q})$

14 In this case, this is a "false alarm".
Galois Connection

- We have got a Galois Connection:
  \[ \langle p(D), \subseteq \rangle \xrightarrow{\alpha} \langle p(D), \subseteq \rangle \]
  \[ \biguparrow \quad \biguparrow \]
  Concrete properties Abstract properties

- With an isomorphic mathematical/computer representation:
  \[ \langle p(D), \subseteq \rangle \xrightarrow{\gamma/\alpha} \langle D^\#, \subseteq \rangle \]
  \[ \biguparrow \quad \biguparrow \]
  Concrete properties Abstract domain

\[ \forall P \in p(D) : \forall Q \in D^\# : \alpha(P) \subseteq Q \quad \iff \quad P \subseteq \gamma(Q) \]

Example 1 de Galois Connection

Example 2 de Galois Connection

Function Abstraction

- Let \( \langle p(D), \subseteq \rangle \xrightarrow{\gamma/\alpha} \langle D^\#, \subseteq \rangle \)
- How to abstract an operator \( F \in p(D) \xrightarrow{\subseteq} p(D) \)?
- The most precise sound overapproximation is
  \[
  F^\# \in D^\# \xrightarrow{\subseteq} D^\#
  \]
  \[
  F^\# = \alpha \circ F \circ \gamma
  \]
- This is a Galois Connection
  \[
  \langle p(D) \xrightarrow{\subseteq} p(D), \subseteq \rangle \xrightarrow{\lambda F^\# \cdot \gamma / \alpha} \langle D^\#, \subseteq \rangle
  \]
Fixpoint Abstraction

- Let \( \langle \wp(D), \subseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}^\|, \subseteq \rangle \)
- How to abstract a fixpoint property \( \text{lfp} \subseteq F \) where \( F \in \wp(D) \xrightarrow{\subseteq} \wp(D) \)?
- Approximate sound abstraction:
  \[ \text{lfp} \subseteq F \subseteq \gamma(\text{lfp} \subseteq \alpha \circ F \circ \gamma) \]
- Complete abstraction: if \( \alpha \circ F = F^\| \circ \alpha \) then
  \[ F^\| = \alpha \circ F \circ \gamma, \quad \alpha(\text{lfp} \subseteq F) = \text{lfp} \subseteq F^\| \]

Example: Accessible States

- Transition system: \( \langle \Sigma, \tau \rangle \)
- Initial states: \( I \subseteq \Sigma \)
- Abstraction: \( \xrightarrow{\alpha} \)
- Accessible states: \( \text{lfp} \subseteq F^\|, \quad F^\| (X) = I \cup \{ s' \mid \exists s \in X : \langle s, s' \rangle \in \tau \} \)

Convergence acceleration of the iterative fixpoint computation

- The fixpoint \( \text{lfp} \subseteq F^\|, \quad F^\| \in \mathcal{D}^\| \xrightarrow{\subseteq} \mathcal{D}^\| \) is computed iteratively\(^{15}\):
  \[ X^0 = \bot \quad X^{n+1} = F^\|(X^n) \quad X^\omega = \bigsqcup_{n \geq 0} X^n \]
- For systems of equations \( \mathcal{D}^\| = \prod_{i=1}^{n} \mathcal{D}^\|_i \), we use asynchronous iterations
- Convergence acceleration techniques have been developed to overapproximate the limit.

Static analysis by abstract interpretation

1. Define the programming language semantics \( S \in \mathcal{L} \mapsto \mathcal{D} \) and the concrete properties \( \wp(D) \);
2. Let \( Q \in \wp(D) \) be the property to be proved about the program \( P : S [P] \in Q \);
3. Choose the abstraction \( \langle \wp(D), \subseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}^\|, \subseteq \rangle \);
4. The abstract interpretation theory formally define an abstract semantics \( S^\| [P] \supseteq \alpha(\{S [P]\}) \);
5. The static analysis algorithm is the computation/overapproximation of the abstract semantics (whence correct by construction).
6. The result of the computation is either
   - \( S \uparrow P \) \( \in \gamma(S \uparrow P) \subseteq Q \) (correctness proof), or
   - \( \gamma(S \uparrow P) \nsubseteq Q \) (property not satisfied (error) or approximation too coarse (false alarm))

7. The abstraction must be chosen in terms of the property \( Q \) to be proved, to be
   - coarse enough to be automatically computable,
   - precise enough to obtain a correctness proof: \( \gamma(S \uparrow P) \subseteq Q \);

Applications of Abstract Interpretation

Any reasoning on complex computer systems must involve a correct approximation of their behaviors, as formalized by Abstract Interpretation [5, 20, 21, 34]

- Syntax of programming languages [30]
- Semantics of programming languages [13, 27]
- Proofs of programs [11, 12]
- Typing and type inference [18]
- Model-checking [23, 28, 31]
- Bisimulations [42]

Abstract interpretation (3) A few applications

- Static analysis of programming languages [3, 7, 15, 16, 22, 26]
  - imperative [2, 4, 6, 9, 19]
  - parallel [10, 8]
  - logic/constraint [14]
  - fonctionnal [17]
- Transformation of programs [29]
- Steganography [33]
- Obfuscation [36]
- Malware detection [37]
- ...
Abstract interpretation (4) Application to critical software

(4.1) The ASTRÉE static analyzer

www.astree.ens.fr [25, 32, 35]

ASTRÉE is a specialized static analyzer

- Embedded **real-time synchronous** control/command C programs:

| Declare and initialize state variables; | Perturbations |
| loop forever | Physical system |
| read volatile input variables, compute output and state variables, write state variables; | Sensors |
| wait for next clock tick end loop | |

Objective of ASTRÉE

- Prove automatically the **absence of runtime errors**:
  - No division by 0, NaN, out of range array access, nil/dangling pointer
  - No signed integer/float overflows
  - Verification of user-defined properties (for example machine dependent properties)

- Requirements:
  - efficiency (must operate on a workstation)
  - precision (few false alarms)

- No alarm → **full certification**
Examples of abstractions

Flooding-point linearization [40, 41]
- Approximate arbitrary expressions in the form
  \[ [a_0, b_0] + \sum_k ([a_k, b_k] \times V_k) \]
- Example:
  \[ Z = X - (0.25 \times X) \]
is linearized as
  \[ Z = ([0.749 \ldots , 0.750 \ldots] \times X) + (2.35 \ldots 10^{-38} \times [-1, 1]) \]
- Allows simplification even in the interval domain
  if \( X \in [-1,1] \), we get \( |Z| \leq 0.750 \ldots \) instead of \( |Z| \leq 1.25 \ldots \)
- Allows using a relational abstract domain (octagons)
- Example of good compromise between cost and precision

General purpose numerical abstract domains

- Intervals: \[ \bigwedge_{i=1}^n a_i \leq x_i \leq b_i \]
- Octagons: \[ \bigwedge_{i,j=1}^n \pm x_i \pm y_j \leq a_{ij} \]
- Polyhedra: \[ \bigwedge_{j=1}^m \left( \sum_{i=1}^n a_{ji} x_i \right) \leq b_j \]

Symbolic abstract domain [40, 41]
- 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 x-x.c
1 void main () { int X, Y;
2   __ASTREE_known_fact(((0 <= X) && (X <= 100)));
3   Y = (X - X);
4   __ASTREE_log_vars((Y));
5 }

astree -exec-fn main -no-relational x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:<interval: Y in [-100, 100]>
astree -exec-fn main x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:<interval: Y in (0) > <symbolic: Y = (X - X)>
```
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 leaves.

---

Ellipsoid Abstract Domain for Filters

- 2nd Order Digital Filter:

```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] */
}
```

```c
void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE;
    }
}
```

---

Control Partitionning for Case Analysis

- Code Sample:

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

### Control point partitionning:

### Trace partitionning:

Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).
Slow divergences by rounding accumulation

- With reals $R$: $x = 1.0$ at $\mathbb{R}$
- With floats: rounding errors
- Accumulation of rounding errors: possible cause of divergence

Solution [35]: bound the cumulated rounding error as a function of the number of iterations by arithmetic-geometric progressions:

- Relation $|x| \leq a \cdot b^n + c$, where $a$, $b$, $c$ are constants determined by the analysis, $n$ is the iterate number
- Number of iterates bounded by $N$: $|x| \leq a \cdot b^N + c$

Arithmetic-geometric progressions $\mathbb{R}^+$

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

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();
    }
}
```

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( ) {
    if (FIRST) { P = X; }
    else {
        P = (P - ((((2.0 * P) - A) - B) * 4.491048e-03));
        B = A;
        if (SWITCH) { A = P; }
        else { A = X; }
    }
}
```

Arithmetic-geometric progressions (Example 2)

```c
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));
```

```c
void main() {
    FIRST = TRUE;
    while (TRUE) {
        dev( );
        FIRST = FALSE;
        __ASTREE_wait_for_clock();
    }
}
```

Potential overflow!

```c
|P| <= (15. + 5.87747175411e-39/ 1.19209290217e-07) * (1+ 1.19209290217e-07)^clock- 5.87747175411e-39 /1.19209290217e-07 <= 23.0393526881
```
(Automatic) Parameterization

- All abstract domains of ASTRËE 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).

Possible origins of imprecision and how to fix it

In case of false alarm, the imprecision can come from:
- Abstract transformers (not best possible) \(\rightarrow\) improve algorithm;
- Automatized parametrization (e.g. variable packing) \(\rightarrow\) improve pattern-matched program schemata;
- Iteration strategy for fixpoints \(\rightarrow\) fix widening \(^{17}\);
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract \(\rightarrow\) 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.

Application to the A 340/A 380

- Primary flight control software of the electric flight control system of the Airbus A340 family and the A380

  \[\text{C program, automatically generated from of high-level specification (à la Simulink/SCADE)}\]
  - A340 : 100.000 to 250.000 LOCs
  - A380 : 400.000 to 1.000.000 LOCs
A world première

- In Nov. 2005, analysis of 400,000 lines of C code\textsuperscript{18}:
  \begin{tabular}{c|c|c}
    \hline
    time & memory & false alarms \\
    \hline
    13h 52mn & 2.2 Gb & 0 \\
    \hline
  \end{tabular}

- In Nov. 2006, analysis of 750,000 lines of C code:
  \begin{tabular}{c|c|c}
    \hline
    time & memory & false alarms \\
    \hline
    34h 30mn & 4.8 Go & 0 \\
    \hline
  \end{tabular}

\textsuperscript{18} on an AMD Opteron 248, 64 bits, a single processor

Static Analysis of Synchronous Programs

- MSU\textsuperscript{19} of the FAS\textsuperscript{20} for the ATV\textsuperscript{21}—ISS\textsuperscript{22} rendezvous (mission critical)
- C version of an ADA program generated from Simulink + Scade + manual code
- 190 000 LOCs

\begin{itemize}
  \item MSU: Monitoring and Safing Unit
  \item FAS: Flight Application Software
  \item ATV: Automated Transfer Vehicle
  \item ISS: International Space Station
\end{itemize}

Current Projects

- Static Analysis of Asynchronous Controlers
  - USB communications
  - Driver + Model of the (hardware) controller
  - Asynchronous interferences between driver/controler\textsuperscript{23}
  - Low-level data structures

\textsuperscript{23} SLAM for example is unsound since it is purely sequential on the driver and completely ignores the controller interfering asynchronously in parallel.
Static Analysis of Asynchronous Programs

- Parallel processes
- Shared variables/semaphors & message communications
- Scheduling with static priorities/delays on waits
- Example application: flight warning system of commercial planes (about 3 500 000 Locs)

Results of the ASTRÉE project

- ASTRÉE is a practical proof that software static analysis by abstract interpretation does scale up
- With a lot of efforts, theoretical & purely speculative research on abstract interpretation can find its way into industrial practice,
- Effective industrial use, if the methodology changes and cost are marginal
- Forthcoming commercialization (1/3 years)

Conclusion

THE END, THANK YOU

More references at URL www.di.ens.fr/~cousot.
<table>
<thead>
<tr>
<th>Bibliographic References</th>
</tr>
</thead>
</table>


