Introduction

MPRI 2–6: Abstract Interpretation, application to verification and static analysis

Antoine Miné

year 2014–2015

course 01
17 September 2014
Motivating program verification
The cost of software failure

- **Patriot MIM-104** failure, 25 February 1991
  (death of 28 soldiers\(^1\))

- **Ariane 5** failure, 4 June 1996
  (cost estimated at more than 370 000 000 US$\(^2\))

- **Toyota** electronic throttle control system failure, 2005
  (at least 89 death\(^3\))

- **Heartbleed** bug in OpenSSL, April 2014

- economic cost of software bugs is tremendous\(^4\)

---

\(^3\) CBSNews. Toyota "Unintended Acceleration" Has Killed 89. 20 March 2014.
Maiden flight of the Ariane 5 Launcher, 4 June 1996.
40s after launch...
**Cause:** software error

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer types

```
P_M_DERIVE(T_ALG.E_BH) :=
    UC_16S_EN_16NS (TDB.T_ENTIER_16S
    ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

- software exception not caught
  \[\implies\] computer switched off

- all backup computers run the same software
  \[\implies\] all computers switched off, no guidance
  \[\implies\] rocket self-destructs

---

5 J.-L. Lions et al., Ariane 501 Inquiry Board report.

How can we avoid such failures?

- Choose a safe programming language.  
  C (low level) / Ada, Java (high level)

- Carefully design the software.  
  many software development methods exist

- Test the software extensively.
How can we avoid such failures?

- Choose a safe programming language.
  - C (low level) / Ada, Java (high level)
  - yet, Ariane 5 software is written in Ada

- Carefully design the software.
  - many software development methods exist
  - yet, critical embedded software follow strict development processes

- Test the software extensively.
  - yet, the erroneous code was well tested... on Ariane 4

⇒ not sufficient!
How can we avoid such failures?

- Choose a safe programming language.
  - C (low level) / Ada, Java (high level)
  - yet, Ariane 5 software is written in Ada

- Carefully design the software.
  - many software development methods exist
  - yet, critical embedded software follow strict development processes

- Test the software extensively.
  - yet, the erroneous code was well tested... on Ariane 4

⇒ **not sufficient!**

We should use **formal methods.**

provide rigorous, mathematical insurance
Proving program properties
Invariants and programs

```plaintext
assume X in [0,1000];

I := 0;

while I < X do
    I := I + 2;

assert I in [0,?]
```

**Goal:** find a bound property, sufficient to express the absence of overflow

---

Invariants and programs

```plaintext
assume X in [0,1000];

I := 0;

while I < X do

    I := I + 2;

assert I in [0,1000]
```

**Goal:** find a bound property, sufficient to express the absence of overflow

---

Invariants and programs

```plaintext
assume X in [0,1000];
{X ∈ [0,1000]}
I := 0;
{X ∈ [0,1000], I = 0}
while I < X do
    {X ∈ [0,1000], I ∈ [0,998]}
    I := I + 2;
    {X ∈ [0,1000], I ∈ [2,1000]}
{X ∈ [0,1000], I ∈ [0,1000]}
assert I in [0,1000]
```

**Invariant**: property true of all the executions of the program

---

Invariants and programs

\begin{verbatim}
assume X in [0,1000];
\{X ∈ [0,1000]\}
I := 0;
\{X ∈ [0,1000], I = 0\}
while I < X do
    \{X ∈ [0,1000], I ∈ [0,998]\}
    I := I + 2;
\{X ∈ [0,1000], I ∈ [2,1000]\}
\{X ∈ [0,1000], I ∈ [0,1000]\}
assert I in [0,1000]
\end{verbatim}

**invariant:** property true of all the executions of the program

**issue:** if \(I = 997\) at a loop iteration, \(I = 999\) at the next

---

Invariants and programs

assume $X$ in $[0,1000]$;
{$X \in [0,1000]$}
$I := 0$;
{$X \in [0,1000], I = 0$}
while $I < X$ do
    {$X \in [0,1000], I \in \{0, 2, \ldots, 996, 998\}$}
    $I := I + 2$;
    {$X \in [0,1000], I \in \{2, 4, \ldots, 998, 1000\}$}
    {$X \in [0,1000], I \in \{0, 2, \ldots, 998, 1000\}$}
assert $I$ in $[0,1000]$

**inductive invariant**: invariant that can be proved to hold by induction on loop iterates
(if $I \in S$ at a loop iteration, then $I \in S$ at the next loop iteration)

---

Robert Floyd

---

Logics and programs

\[
\begin{align*}
\{P[e/X]\} & \quad X := e \{P\} \\
\{P\} C_1 \{R\} \quad \{R\} C_2 \{Q\} & \quad \{P\} C_1; C_2 \{Q\} \\
\{P \& b\} & \quad C \{P\} \\
\{P\} & \quad \text{while } b \text{ do } C \{P \& \neg b\} \\
\end{align*}
\]

- sound logic to prove program properties, (rel.) complete
- proofs can be partially automated (at least proof checking)
  (e.g., using proof assistants: Coq, PVS, Isabelle, HOL, etc.)

---

sound logic to prove program properties, (rel.) complete

proofs can be partially automated (at least proof checking)
(e.g., using proof assistants: Coq, PVS, Isabelle, HOL, etc.)

requires annotations and interaction with a prover
even manual annotation is not practical for large programs

---

Limit to automation
The computer $O$ runs the program $P$ on the data $D$ and answers $(yes, no)$... or does not answer ($\perp$).
$O(P, D) \in \{\text{yes}, \text{no}, \bot\}$

Note that programs are also a kind of data! They can be fed to other programs. (e.g., to compilers)
Static analyzer $A$.

Given a program $P$:

- $O(A, P) = \text{yes} \iff \forall D, O(P, D) \text{ is safe}$
- $O(A, P) \neq \bot$ (the static analysis always terminates)
Static analysis

Static analyzer $A$.

Given a program $P$:

- $O(A, P) = \text{yes} \iff \forall D, O(P, D) \text{ is safe}
- O(A, P) \neq \bot$ \quad (the static analysis always terminates)

$\implies P$ is proved safe even before it is run!
There cannot exist a static analyzer $A$ proving the termination of every terminating program $P$.

---


Fundamental undecidability

There cannot exist a static analyzer $A$ proving the termination of every terminating program $P$.

Proof sketch:

$A(P \cdot D): O(A, P \cdot D) = \begin{cases} yes & \text{if } O(P, D) \neq \bot \\ no & \text{otherwise} \end{cases}$

$A'(X):$ while $A(X \cdot X)$ do nothing; no

do we have $O(A', A') = \bot$ or $\neq \bot$? neither!
$
\implies A$ cannot exist

All “interesting” properties are undecidable!

---


Approximation
An approximate static analyzer $A$ always answers in finite time ($\neq \bot$):

- either **yes**: the program $P$ is definitely safe (soundness)
- either **no**: I don’t know (incompleteness)

Sufficient to prove the safety of (some) programs. Fails on infinitely many programs...
An approximate static analyzer $A$ always answers in finite time ($\neq \perp$):

- either \textit{yes}: the program $P$ is definitely safe \hspace{1cm} (soundness)
- either \textit{no}: I don’t know \hspace{1cm} (incompleteness)

Sufficient to prove the safety of (some) programs. Fails on infinitely many programs.

$\implies$ We should \textit{adapt} the analyzer $A$ to

- a class of programs to verify, and
- a class of safety properties to check.
Abstract interpretation

Patrick Cousot

General theory of the approximation and comparison of program semantics:

- unifies many existing semantics
- allows the definition of new static analyses that are correct by construction

Abstract interpretation

\((S_0)\)
assume \(X\) in \([0,1000]\);

\((S_1)\)
\(I := 0;\)

\((S_2)\)
while \((S_3)\) \(I < X\) do

\((S_4)\)
\(I := I + 2;\)

\((S_5)\)

\((S_6)\)
program
Abstract interpretation

\[ (S_0) \]
assume \( X \) in \([0,1000]\);

\[ (S_1) \]
\( I := 0; \)

\[ (S_2) \]
while \((S_3)\) \( I < X \) do

\[ (S_4) \]
\( I := I + 2; \)

\[ (S_5) \]

\[ (S_6) \]
program semantics

Concrete semantics \( S_i \in \mathcal{D} = \mathcal{P}(\{I, X\} \to \mathbb{Z}) \):

- strongest invariant (and an inductive invariant)
- not computable in general
- smallest solution of a system of equations
Abstract interpretation

$$(S_0)$$

assume X in [0,1000];

$$(S_1)$$

I := 0;

$$(S_2)$$

while $$(S_3)$$ I < X do

$$(S_4)$$

I := I + 2;

$$(S_5)$$

program semantics

Abstract semantics $S_i^\# \in D^#$:

- $D^#$ is a subset of properties of interest (approximation)
with a machine representation

- $F^# : D^# \rightarrow D^#$ over-approximates the effect of $F : D \rightarrow D$ in $D^#$
(with effective algorithms)
Numeric abstract domain examples

concrete sets $D$: $\{(0, 3), (5.5, 0), (12, 7), \ldots\}$
Numeric abstract domain examples

concrete sets $\mathcal{D}$: $\{(0, 3), (5.5, 0), (12, 7), \ldots \}$
abstract polyhedra $\mathcal{D}_p^\#$: $6X + 11Y \geq 33 \wedge \cdots$
Numeric abstract domain examples

concrete sets $D$: $\{(0, 3), (5.5, 0), (12, 7), \ldots\}$
abstract polyhedra $D^\#_p$: $6X + 11Y \geq 33 \land \cdots$
abstract octagons $D^\#_o$: $X + Y \geq 3 \land Y \geq 0 \land \cdots$
numeric abstract domain examples

concrete sets $\mathcal{D}$:  
$\{(0, 3), (5.5, 0), (12, 7), \ldots \}$

abstract polyhedra $\mathcal{D}^\#_p$:  
$6X + 11Y \geq 33 \land \cdots$

abstract octagons $\mathcal{D}^\#_o$:  
$X + Y \geq 3 \land Y \geq 0 \land \cdots$

abstract intervals $\mathcal{D}^\#_i$:  
$X \in [0, 12] \land Y \in [0, 8]$
Numeric abstract domain examples

concrete sets $\mathcal{D}$: \{(0, 3), (5.5, 0), (12, 7), \ldots\} not computable
abstract polyhedra $\mathcal{D}_p^\#$: $6X + 11Y \geq 33 \land \cdots$ exponential cost
abstract octagons $\mathcal{D}_o^\#$: $X + Y \geq 3 \land Y \geq 0 \land \cdots$ cubic cost
abstract intervals $\mathcal{D}_i^\#$: $X \in [0, 12] \land Y \in [0, 8]$ linear cost

Trade-off between cost and expressiveness / precision
The program is correct \((\text{blue} \cap \text{red} = \emptyset)\).
The program is correct \((\text{blue} \cap \text{red} = \emptyset)\).
The polyhedra domain can prove the correctness \((\text{cyan} \cap \text{red} = \emptyset)\).
The program is **correct** (blue ∩ red = ∅).
The polyhedra domain can **prove the correctness** (cyan ∩ red = ∅).
The interval domain **cannot** (green ∩ red ≠ ∅, false alarm).
abstract semantics $F^\#$ in the interval domain $\mathcal{D}_i^\#
$
- $I \in \mathcal{D}_i^\#$ is a pair of bounds $(\ell, h) \in \mathbb{Z}^2$ (for each variable) representing an interval $[\ell, h] \subseteq \mathbb{Z}$
- $I := I + 2$: $(\ell, h) \mapsto (\ell + 2, h + 2)$
- $\cup^\#: (\ell_1, h_1) \cup^\# (\ell_2, h_2) = (\min(\ell_1, \ell_2), \max(h_1, h_2))$
- $\ldots$
Resolution by iteration and extrapolation

**Challenge:** the equation system is recursive: $\mathbf{S}^\# = \mathbf{F}^\#(\mathbf{S}^\#)$.

**Solution:** resolution by iteration: $\mathbf{S}^\#_0 = \emptyset^\#, \mathbf{S}^\#_{i+1} = \mathbf{F}^\#(\mathbf{S}^\#_i)$.

e.g., $\mathbf{S}^\#_3$: $I \in \emptyset$, $I = 0$, $I \in [0, 2]$, $I \in [0, 4]$, $\ldots$, $I \in [0, 1000]$. 


Resolution by iteration and extrapolation

**Challenge:** the equation system is recursive: \( \vec{S}^\# = \vec{F}^\#(\vec{S}^\#) \).

**Solution:** resolution by iteration: \( \vec{S}^{\#}_0 = \emptyset^\#, \vec{S}^{\#}_{i+1} = \vec{F}^\#(\vec{S}^{\#}_i) \).

e.g., \( S^\#_3 : I \in \emptyset, \ I = 0, \ I \in [0, 2], \ I \in [0, 4], \ldots, \ I \in [0, 1000] \)

**Challenge:** infinite or very long sequence of iterates in \( D^\# \)

**Solution:** extrapolation operator \( \nabla \)

e.g., \( [0, 2] \nabla [0, 4] = [0, +\infty[ \)

- remove unstable bounds and constraints
- ensures the convergence in finite time
- **inductive reasoning** (through generalisation)
Resolution by iteration and extrapolation

**Challenge:** the equation system is recursive: $\vec{S}^\# = \vec{F}^\#(\vec{S}^\#)$.

**Solution:** resolution by iteration: $\vec{S}^\#_0 = \emptyset^\#, \vec{S}^\#_{i+1} = \vec{F}^\#(\vec{S}^\#_i)$.  

e.g., $S^\#_3 : I \in \emptyset, I = 0, I \in [0, 2], I \in [0, 4], \ldots, I \in [0, 1000]$

**Challenge:** infinite or very long sequence of iterates in $D^\#$

**Solution:** extrapolation operator $\triangleleft$

e.g., $[0, 2] \triangleleft [0, 4] = [0, +\infty]$

- remove unstable bounds and constraints
- ensures the convergence in finite time
- inductive reasoning (through generalisation)

$\implies$ effective solving method $\implies$ static analyzer!
Other uses of abstract interpretation

- Analysis of dynamic memory data-structures (*shape analysis*).
- Analysis of parallel, distributed, and multi-thread programs.
- Analysis of probabilistic programs.
- Analysis of biological systems.
- Security analysis (*information flow*).
- Termination analysis.
- Cost analysis.
- Analyses to enable compiler optimisations.
- ...
Some static analysis tools based on Abstract Interpretation
The Astrée static analyzer
The Astrée static analyzer

Analyseur statique de programmes temps-réels embarqués
(static analyzer for real-time embedded software)

- developed at ENS
  B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, D. Monniaux, A. Miné, X. Rival

- industrialized and made commercially available by AbsInt

Astrée
www.astree.ens.fr

AbsInt
www.absint.com
The Astrée static analyzer

Specialized:

- for the analysis of run-time errors
  (arithmetic overflows, array overflows, divisions by 0, etc.)

- on embedded critical C software
  (no dynamic memory allocation, no recursivity)

- in particular on control / command software
  (reactive programs, intensive floating-point computations)

- intended for validation
  (analysis does not miss any error and tries to minimise false alarms)
The Astrée static analyzer

Specialized:

- for the analysis of run-time errors
  (arithmetic overflows, array overflows, divisions by 0, etc.)

- on embedded critical C software
  (no dynamic memory allocation, no recursivity)

- in particular on control / command software
  (reactive programs, intensive floating-point computations)

- intended for validation
  (analysis does not miss any error and tries to minimise false alarms)

Approximately 40 abstract domains are used at the same time:

- numeric domains (intervals, octagons, ellipsoids, etc.)
- boolean domains
- domains expressing properties on the history of computations
Astrée applications


- size: from 70 000 to 860 000 lines of C
- analysis time: from 45mn to $\approx 40h$
- 0 alarm: proof of absence of run-time error

```c
#include "daed_builtins.h"
int main() {
    int i;
    double y = 0.7;
    double x = y;
    for (i = 1; i <= 20; i++) {
        x = 11 * x - 7;
    }
    return 0;
}
```
Static analysis of the **accuracy of floating-point computations**:

- bound the range of variables
- bound the **rounding errors** wrt. real computation
- track the **origin** of rounding errors
  (which operation contributes to most error, target for improvements)
- uses specific abstract domains
  (affine arithmetic, zonotopes)

- developed at **CEA-LIST** (E. Goubault, S. Putot)
- industrial use (Airbus)
Clousot: CodeContract static checker
Clousot: CodeContract static checker

**CodeContracts:**

- **assertion** language for .NET (C#, VB, etc.)
  (pre-conditions, post-conditions, invariants)

- dynamic checking
  (insert run-time checks)

- **static checking**
  (modular abstract interpretation)

- **automatic inference**
  (abstract interpretation to infer necessary preconditions backwards)

- developed at Microsoft Research *(M. Fahndrich, F. Logozzo)*
- part of .NET Framework 4.0
- integrated to Visual Studio