Introduction

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

Antoine Miné

year 2016–2017

course 01b
14 September 2016

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]

---

[1] R. Skeel. "Roundoff Error and the Patriot Missile". SIAM News, volume 25, nr 4.

[2] M. Dowson. "The Ariane 5 Software Failure". Software Engineering Notes 22 (2): 84, March 1997.

[3] CBSNews. Toyota "Unintended Acceleration" Has Killed 89. 20 March 2014.

[4] NIST. Software errors cost U.S. economy $59.5 billion annually. Tech. report, NIST Planning Report, 2002.

Zoom on: Ariane 5, Flight 501

Maiden flight of the Ariane 5 Launcher, 4 June 1996.

Zoom on: Ariane 5, Flight 501

40s after launch. . .

Zoom on: Ariane 5, Flight 501

Cause: software error[5]

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer types[6]

```
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
  $\Longrightarrow$ computer switched off
- all backup computers run the same software
  $\Longrightarrow$ all computers switched off, no guidance
  $\Longrightarrow$ rocket self-destructs

---

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

[6] J.-J. Levy. Un petit bogue, un grand boum. Séminaire du Département d'informatique de l'ENS, 2010.

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

$\Longrightarrow$ **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

$\Longrightarrow$ **not sufficient!**

We should use **formal methods.**

provide rigorous, mathematical insurance

Proving program properties

Invariants and programs

```
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

---

[7] R. W. Floyd. "Assigning meanings to programs". In Proc. Amer. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19–31, 1967.

Invariants and programs

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

I := 0;

while I < X do

    I := I + 2;


assert I in [0,1000]
```

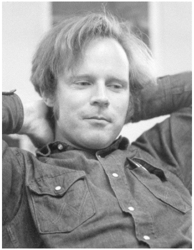
Goal: find a bound property, sufficient to express the absence of overflow

---

[7] R. W. Floyd. "Assigning meanings to programs". In Proc. Amer. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19–31, 1967.

Invariants and programs

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



Robert Floyd[7]

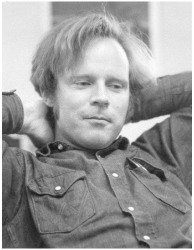
invariant: property true of all the executions of the program

---

[7] R. W. Floyd. "Assigning meanings to programs". In Proc. Amer. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19–31, 1967.

Invariants and programs

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



Robert Floyd[7]

invariant: property true of all the executions of the program

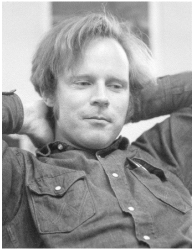
issue: if $I = 997$ at a loop iteration, $I = 999$ at the next

---

[7] R. W. Floyd. "Assigning meanings to programs". In Proc. Amer. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19–31, 1967.

Invariants and programs

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



Robert Floyd[7]

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)

---

[7] R. W. Floyd. "Assigning meanings to programs". In Proc. Amer. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19–31, 1967.

Logics and programs

$$\frac{}{\{P[e/X]\}\ \mathtt{X := e}\ \{P\}} \qquad \frac{\{P\}\ \mathtt{C_1}\ \{R\} \quad \{R\}\ \mathtt{C_2}\ \{Q\}}{\{P\}\ \mathtt{C_1;C_2}\ \{Q\}}$$

$$\frac{\{P \,\&\, b\}\ \mathtt{C}\ \{P\}}{\{P\}\ \mathtt{while\ b\ do\ C}\ \{P \,\&\, \neg b\}}$$

$$\ldots$$

Tony Hoare[8]

- 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.)

---

[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". Commun. ACM 12(10): 576–580 (1969).

Logics and programs

$$\frac{}{\{P[e/X]\}\ \texttt{X := e}\ \{P\}} \qquad \frac{\{P\}\ \texttt{C}_1\ \{R\} \quad \{R\}\ \texttt{C}_2\ \{Q\}}{\{P\}\ \texttt{C}_1;\texttt{C}_2\ \{Q\}}$$

$$\frac{\{P \,\&\, b\}\ \texttt{C}\ \{P\}}{\{P\}\ \texttt{while b do C}\ \{P \,\&\, \neg b\}}$$

$$\ldots$$

Tony Hoare[8]

- 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

---

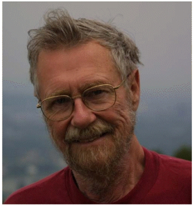
[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". Commun. ACM 12(10): 576–580 (1969).

A calculs of program properties

$$wlp(\mathtt{X} := \mathtt{e}, P) \overset{\text{def}}{=} P[e/X]$$

$$wlp(\mathtt{C}_1; \mathtt{C}_2, P) \overset{\text{def}}{=} wlp(\mathtt{C}_1, wlp(\mathtt{C}_2, P))$$

$$wlp(\mathtt{while}\ \mathtt{e}\ \mathtt{do}\ \mathtt{C}, P) \overset{\text{def}}{=}$$
$$I \land ((e \land I) \implies wlp(\mathtt{C}, I)) \land ((\neg e \land I) \implies P)$$



Edsger W. Dijkstra[9]

- predicate transformer semantics
  propagate predicates on states through the program
- weakest (liberal) precondition
  backwards, from property to condition for program correctness
- calculs that can be mostly automated

---

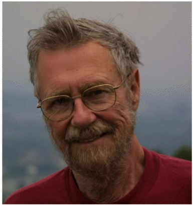
[9] E. W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". EWD472. Commun. ACM 18(8): 453-457 (1975).

A calculs of program properties

$$wlp(\texttt{X} := \texttt{e}, P) \overset{\text{def}}{=} P[e/X]$$

$$wlp(\texttt{C}_1;\texttt{C}_2, P) \overset{\text{def}}{=} wlp(\texttt{C}_1, wlp(\texttt{C}_2, P))$$

$$wlp(\texttt{while e do C}, P) \overset{\text{def}}{=}$$
$$I \land ((e \land I) \implies wlp(\texttt{C}, I)) \land ((\neg e \land I) \implies P)$$



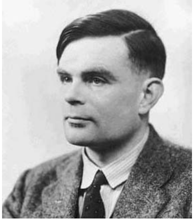
Edsger W. Dijkstra[9]

- predicate transformer semantics
  propagate predicates on states through the program
- weakest (liberal) precondition
  backwards, from property to condition for program correctness
- calculs that can be mostly automated, except for:
  - user annotations for inductive loop invariants
  - function annotations (modular inference)
- academic success: complex (functional) but local properties
- industry success: for simple, local properties

---

[9] E. W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". EWD472. Commun. ACM 18(8): 453-457 (1975).

Static analysis



Alan Turing

Principle: a program A that

- takes as input another program P
  (programs are also data!)
- answers with "yes" if the program is safe,
  "no" if it is not safe
- always answers, hopefully quickly

$\Longrightarrow$ proves automatically a program safe before it is run!

Limit to automation: undecidability

It is well known that termination (a useful property) is undecidable.[10]

In fact, all "interesting" properties are undecidable[11]

$\Longrightarrow$ A cannot exist. ☹

---

[10] A. M. Turing. "Computability and definability". The Journal of Symbolic Logic, vol. 2, pp. 153–163, (1937).

[11] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems." Trans. Amer. Math. Soc. 74, 358-366, 1953.

Abstract interpretation

Approximate static analysis

An approximate static analyzer *A* always answers in finite time

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

Sufficient to prove the safety of (some) programs.

Incompleteness: *A* fails on infinitely many programs. . .

Approximate static analysis

An approximate static analyzer A always answers in finite time

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

Sufficient to prove the safety of (some) programs.

Incompleteness: A fails on infinitely many programs. . .

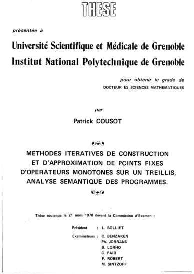
Completeness: for any safe program P, we can design an analyzer A that proves it!

$\Longrightarrow$ We should adapt the analyzer A to

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

Abstract interpretation

Patrick Cousot[12]



THESE

présentée à

Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR ES SCIENCES MATHEMATIQUES

par

Patrick COUSOT

METHODES ITERATIVES DE CONSTRUCTION
ET D'APPROXIMATION DE POINTS FIXES
D'OPERATEURS MONOTONES SUR UN TREILLIS,
ANALYSE SEMANTIQUE DES PROGRAMMES.

Thèse soutenue le 21 mars 1978 devant la Commission d'Examen :

Président : L. BOLLIET
Examinateurs : C. BENZAKEN
Ph. JORRAND
B. LORHO
C. PAIR
F. ROBERT
M. SINTZOFF

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

---

[12] P. Cousot. "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes." Thèse És Sciences Mathématiques, 1978.

Abstract interpretation: LIP6 Colloquium

Colloquium d'Informatique de l'UPMC Sorbonne Universités

contact : colloquium@lip6.fr
http://www.lip6.fr/colloquium/
Vidéo disponible sur le site

Abstract interpretation

Patrick Cousot
New York University

Amphi 15

4, place Jussieu
75005 Paris
Metro Jussieu

29 Septembre 2016
à 18h00

The complexity of large programs grows faster than the intellectual ability of programmers in charge of their development and maintenance. The direct consequence is a lot of errors and bugs in programs mostly debugged by their end-users. Programmers are not responsible for these bugs. They are not required to produce provably safe and secure programs. This is because professionals are only required to apply state of the art techniques, that is testing on finitely many cases. This state of the art is changing rapidly and so will irresponsibility, as in other manufacturing disciples.

Scalable and cost-effective tools have appeared recently that can avoid bugs with possible dramatic consequences for example in transportation, banks, privacy of social networks, etc. Entirely automatic, they are able to capture all bugs involving the violation of software healthiness rules such as the use of operations with arguments for which they are undefined.

These tools are formally founded on abstract interpretation. They are based on a definition of the semantics of programming languages specifying all possible executions of the programs of a language. Program properties of interest are abstractions of these semantics abstracting away all aspects of the semantics not relevant to a particular reasoning on programs. This yields proof methods.

Full automation is more difficult because of undecidability: programs cannot always prove programs correct in finite time and memory. Further abstractions are therefore necessary for automation, which introduce imprecision. Bugs may be signalled that are impossible in any execution (but still none is forgotten). This has an economic cost, much less than testing. Moreover, the best static analysis tools are able to reduce these false alarms to almost zero. A time-consuming and error-prone task which is too difficult, if not impossible for programmers, without tools.

Patrick Cousot received the Doctor Engineer degree in Computer Science and the Doctor ès Sciences degree in Mathematics from the University Joseph Fourier of Grenoble, France. He was a Research Scientist at the French National Center for Scientific Research at the University Joseph Fourier of Grenoble, France, then professor at the University of Metz, the École Polytechnique, the École Normale Supérieure, Paris, France. He is Silver Professor of Computer Science at the Courant Institute of Mathematical Sciences, New York University, USA. Patrick Cousot is the inventor, with Radhia Cousot, of Abstract Interpretation.

Talk by Patrick Cousot at Paris 6, 29 September 2016, 18h00
https://www.lip6.fr/colloquium/

Abstract interpretation

```
$(\mathcal{S}_0)$
assume X in [0,1000];
$(\mathcal{S}_1)$
I := 0;
$(\mathcal{S}_2)$
while $(\mathcal{S}_3)$ I < X do
    $(\mathcal{S}_4)$
    I := I + 2;
    $(\mathcal{S}_5)$
$(\mathcal{S}_6)$
```

program

Abstract interpretation

```
(S₀)
assume X in [0,1000];
(S₁)
I := 0;
(S₂)
while (S₃) I < X do
    (S₄)
    I := I + 2;
    (S₅)
(S₆)
```

program

$$\mathcal{S}_i \in \mathcal{D} = \mathcal{P}(\{\mathtt{I},\mathtt{X}\} \to \mathbb{Z})$$

$$\begin{array}{lll}
\mathcal{S}_0 = \{\,(i,x) \mid i,x \in \mathbb{Z}\,\} & = \top \\
\mathcal{S}_1 = \{\,(i,x) \in \mathcal{S}_0 \mid x \in [0,1000]\,\} & = F_1(\mathcal{S}_0) \\
\mathcal{S}_2 = \{\,(0,x) \mid \exists i,\ (i,x) \in \mathcal{S}_1\,\} & = F_2(\mathcal{S}_1) \\
\mathcal{S}_3 = \mathcal{S}_2 \cup \mathcal{S}_5 & \\
\mathcal{S}_4 = \{\,(i,x) \in \mathcal{S}_3 \mid i < x\,\} & = F_4(\mathcal{S}_3) \\
\mathcal{S}_5 = \{\,(i+2,x) \mid (i,x) \in \mathcal{S}_4\,\} & = F_5(\mathcal{S}_4) \\
\mathcal{S}_6 = \{\,(i,x) \in \mathcal{S}_3 \mid i \geq x\,\} & = F_6(\mathcal{S}_3)
\end{array}$$

semantics

Concrete semantics $\mathcal{S}_i \in \mathcal{D} = \mathcal{P}(\{\mathtt{I},\mathtt{X}\} \to \mathbb{Z})$:

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

Abstract interpretation

```
$(\mathcal{S}_0)$
assume X in [0,1000];
$(\mathcal{S}_1)$
I := 0;
$(\mathcal{S}_2)$
while $(\mathcal{S}_3)$ I < X do
    $(\mathcal{S}_4)$
    I := I + 2;
    $(\mathcal{S}_5)$
$(\mathcal{S}_6)$
```

program

$$\mathcal{S}_i^\sharp \in \mathcal{D}^\sharp$$

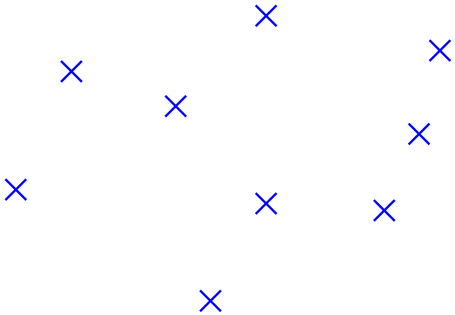
$$\begin{aligned}
\mathcal{S}_0^\sharp &= \top^\sharp \\
\mathcal{S}_1^\sharp &= [\![\, \textit{assume } X \in [0,1000] \,]\!]^\sharp(\mathcal{S}_0^\sharp) \\
\mathcal{S}_2^\sharp &= [\![\, I \leftarrow 0 \,]\!]^\sharp(\mathcal{S}_1^\sharp) \\
\mathcal{S}_3^\sharp &= S_2^\sharp \cup^\sharp \mathcal{S}_5^\sharp \\
\mathcal{S}_4^\sharp &= [\![\, \textit{assume } I < X \,]\!]^\sharp(\mathcal{S}_3^\sharp) \\
\mathcal{S}_5^\sharp &= [\![\, I \leftarrow I + 2 \,]\!]^\sharp(\mathcal{S}_4^\sharp) \\
\mathcal{S}_6^\sharp &= [\![\, \textit{assume } I \geq X \,]\!]^\sharp(\mathcal{S}_3^\sharp)
\end{aligned}$$

semantics

Abstract semantics $\mathcal{S}_i^\sharp \in \mathcal{D}^\sharp$:

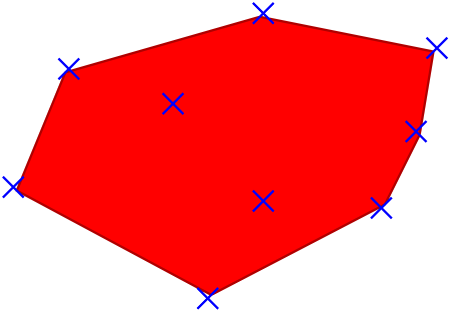
- $\mathcal{D}^\sharp$ is a subset of properties of interest (approximation)
  with a machine representation
- $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ over-approximates the effect of $F : \mathcal{D} \to \mathcal{D}$ in $\mathcal{D}^\sharp$
  (with effective algorithms)

Numeric abstract domain examples



concrete sets: $\{(0,3),(5.5,0),(12,7),\ldots\} \subseteq \mathbb{R}^2$

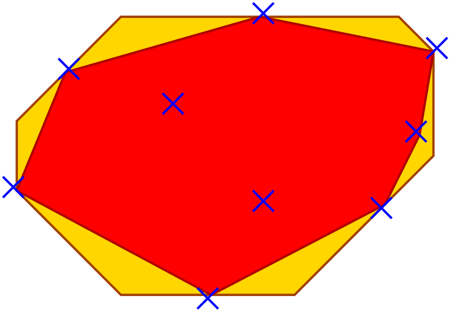
Numeric abstract domain examples



concrete sets: $\{(0,3),(5.5,0),(12,7),\ldots\} \subseteq \mathbb{R}^2$

abstract polyhedra: $6X + 11Y \geq 33 \wedge \cdots$

Numeric abstract domain examples

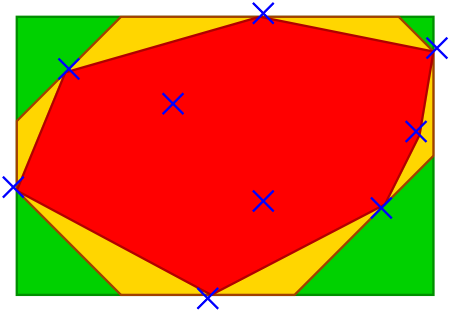


concrete sets: $\{(0,3),(5.5,0),(12,7),\ldots\} \subseteq \mathbb{R}^2$

abstract polyhedra: $6X + 11Y \geq 33 \wedge \cdots$

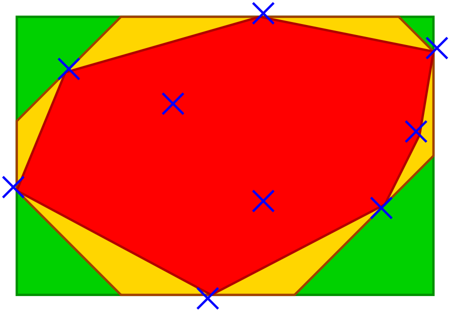
abstract octagons: $X + Y \geq 3 \wedge Y \geq 0 \wedge \cdots$

Numeric abstract domain examples



concrete sets:	$\{(0,3),(5.5,0),(12,7),\ldots\} \subseteq \mathbb{R}^2$
abstract polyhedra:	$6X + 11Y \geq 33 \wedge \cdots$
abstract octagons:	$X + Y \geq 3 \wedge Y \geq 0 \wedge \cdots$
abstract intervals:	$X \in [0,12] \wedge Y \in [0,8]$

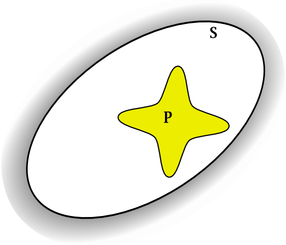
Numeric abstract domain examples



concrete sets:	$\{(0,3),(5.5,0),(12,7),\ldots\} \subseteq \mathbb{R}^2$	not computable
abstract polyhedra:	$6X + 11Y \geq 33 \wedge \cdots$	exponential cost
abstract octagons:	$X + Y \geq 3 \wedge Y \geq 0 \wedge \cdots$	cubic cost
abstract intervals:	$X \in [0,12] \wedge Y \in [0,8]$	linear cost

Trade-off between cost and expressiveness / precision

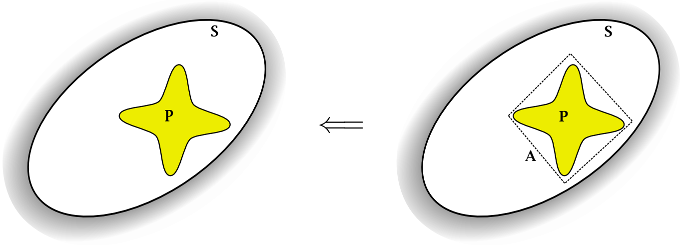
Soundness and false alarms



$$P \subseteq S$$

Goal : prove that a program P satisfies its specification S

Soundness and false alarms



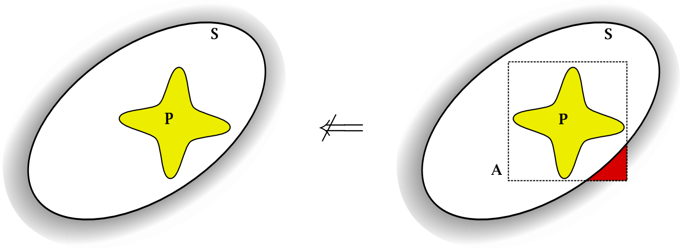
$P \subseteq S$

$A \subseteq S$
program proved

<u>Goal :</u> prove that a program P satisfies its specification S

A polyhedral abstraction A can prove the correctness.

Soundness and false alarms



$P \subseteq S$ $\not\Longleftarrow$ $A \not\subseteq S$

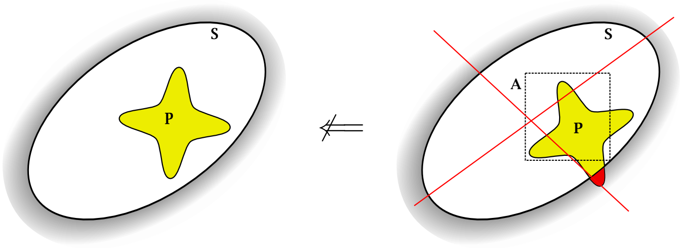
false alarm

<u>Goal :</u> prove that a program P satisfies its specification S

A polyhedral abstraction A can prove the correctness.

An interval abstraction cannot prove the correctness
$\Longrightarrow$ false alarm.

Soundness and false alarms



$P \subseteq S$

cannot occur
false negative

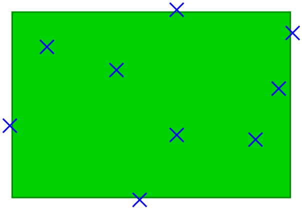
Goal : prove that a program P satisfies its specification S

A polyhedral abstraction A can prove the correctness.

An interval abstraction cannot prove the correctness
$\Longrightarrow$ false alarm.

The analaysis is sound: no false negative reported!

Abstract elements and operators



abstract semantics $F^\sharp$ in the interval domain $\mathcal{D}_i^\sharp$

- $I \in \mathcal{D}_i^\sharp$ 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`: $(\ell, h) \mapsto (\ell+2, h+2)$
- $\cup^\sharp$: $(\ell_1, h_1) \cup^\sharp (\ell_2, h_2) = (\min(\ell_1, \ell_2), \max(h_1, h_2))$
- ...

Resolution by iteration and extrapolation

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

Solution: resolution by iteration: $\vec{\mathcal{S}}^{\sharp\,0} = \emptyset^\sharp$, $\vec{\mathcal{S}}^{\sharp\,i+1} = \vec{F}^\sharp(\vec{\mathcal{S}}^{\sharp\,i})$.

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

Resolution by iteration and extrapolation

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

Solution: resolution by iteration: $\vec{\mathcal{S}}^{\sharp\,0} = \emptyset^\sharp$, $\vec{\mathcal{S}}^{\sharp\,i+1} = \vec{F}^\sharp(\vec{\mathcal{S}}^{\sharp\,i})$.

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

Challenge: infinite or very long sequence of iterates in $\mathcal{D}^\sharp$

Solution: extrapolation operator $\triangledown$

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

- 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: $\vec{\mathcal{S}}^\sharp = \vec{F}^\sharp(\vec{\mathcal{S}}^\sharp)$.

Solution: resolution by iteration: $\vec{\mathcal{S}}^{\sharp\,0} = \emptyset^\sharp$, $\vec{\mathcal{S}}^{\sharp\,i+1} = \vec{F}^\sharp(\vec{\mathcal{S}}^{\sharp\,i})$.

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

Challenge: infinite or very long sequence of iterates in $\mathcal{D}^\sharp$

Solution: extrapolation operator $\triangledown$

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

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

$\Longrightarrow$ effective solving method $\longrightarrow$ static analyzer!

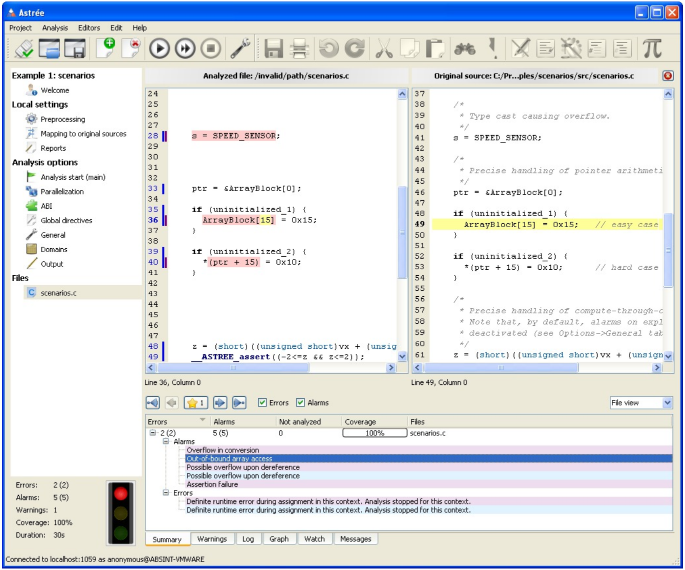
Other uses of abstract interpretation

- 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.
- ...

A few examples of abstract interpretation tools

- Proprietary tools
  - PolySpace analyzer *(MathWorks)*
    run-time errors in Ada, C, C++
  - aiT *(AbsInt)*
    worst-case execution time for binary
  - Astrée *(CNRS, ENS, INRIA, AbsInt)*
    run-time errors in embedded C, with an emphasis on validation
  - Sparrow *(Seoul National University)*
    run-time errors in C
  - Julia *(University of Verona)*
    analysis of Java and Andorid
- Open-source tools
  - Frama-C *(CEA LIST, INRIA, TrustInSoft)*
    run-time errors in C software, also has a commercial version
  - Code Contracts Static Checker *(Microsoft Research)*
    static checking and inference of .NET contracts

The Astrée static analyzer



Analyzed file: /invalid/path/scenarios.c

```c




  s = SPEED_SENSOR;




  ptr = &ArrayBlock[0];

  if (uninitialized_1) {
    ArrayBlock[15] = 0x15;
  }

  if (uninitialized_2) {
    *(ptr + 15) = 0x10;
  }






  z = (short)((unsigned short)vx + (unsig
  __ASTREE_assert((-2<=z && z<=2));
```

Line 36, Column 0

Original source: C:/Pr...ples/scenarios/src/scenarios.c

```c

  /*
   * Type cast causing overflow.
   */
  s = SPEED_SENSOR;

  /*
   * Precise handling of pointer arithmeti
   */
  ptr = &ArrayBlock[0];

  if (uninitialized_1) {
    ArrayBlock[15] = 0x15;    // easy case
  }

  if (uninitialized_2) {
    *(ptr + 15) = 0x10;       // hard case
  }

  /*
   * Precise handling of compute-through-c
   * Note that, by default, alarms on expl
   * deactivated (see Options->General tak
   */
  z = (short)((unsigned short)vx + (unsign
```

Line 49, Column 0

Errors	Alarms	Not analyzed	Coverage	Files
2 (2)	5 (5)	0	100%	scenarios.c

```
Alarms
    Overflow in conversion
    Out-of-bound array access
    Possible overflow upon dereference
    Possible overflow upon dereference
    Assertion failure
Errors
    Definite runtime error during assignment in this context. Analysis stopped for this context.
    Definite runtime error during assignment in this context. Analysis stopped for this context.
```

Errors: 2 (2)
Alarms: 5 (5)
Warnings: 1
Coverage: 100%
Duration: 30s

Connected to localhost:1059 as anonymous@ABSINT-VMWARE

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)

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

Airbus A340-300 (2003)

Airbus A380 (2004)

(model of) ESA ATV (2008)

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