Introduction

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

Antoine Miné

Year 2018–2019

Course 00
11 September 2018
Motivation
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

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

---

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

Zoom on: Ariane 5, Flight 501

**Cause:** software error

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

  \[
  \text{P.M.DERIVE(T.ALG.E.BH)} := \\
  \text{UC16S_EN16NS (TDB.T.ENTIER16S (}}(1.0/C.M.LSB_BH) * \text{G.M.INFO.DERIVE(T.ALG.E.BH))});
  \]

- software **exception not caught**
  \[\Rightarrow\text{ computer switched off}\]
- all backup computers run the same software
  \[\Rightarrow\text{ all computers switched off, no guidance}\]
  \[\Rightarrow\text{ rocket self-destructs}\]

A “simple” error…

---

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, OCaml (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, OCaml (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, OCaml (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 of correctness
may not prove everything, but give a precise notion of what is proved
Verification: compromises

**Undecidability**: correctness properties are undecidable!
cannot build a program that automatically and precisely separates all correct programs from all incorrect ones

Possible compromises:
lose automation, or completeness, or soundness, or generality, or all

- **Test**: complete and automatic, but unsound
- **Theorem proving**
  - proof essentially manual, but checked automatically
  - powerful, but very steep learning curve
- **Deductive methods**
  - automated proofs for some logic fragments (SAT, SMT)
  - still requires program annotations (contracts, invariants)
- **Model checking**
  - check a (often hand-crafted) model of the program
  - finite or regular models, expressive properties (LTL)
  - automatic and complete (wrt. model)
Verification by static analysis

- work directly on the **source code**
- infer properties on **program executions**
- **automatically** (cost effective)
- construct dynamically a **semantic abstraction** of the program
- deduce program **correctness** or raise **alarms**
  (implicit specification: absence of RTE; or user-defined properties: contracts)
- with **approximations** (incomplete: efficient, but possible false alarms)
- **soundly** (no false positive)
Critical avionics software is subject to certification:

- more than half the development cost
- regulated by international standards (DO-178B, DO-178C)
- mostly based on massive test campaigns & intellectual reviews

**Current trend:**

use of **formal methods** now acknowledged (DO-178C, DO-333)

- at the binary level, to replace testing
- at the **source level**, to replace intellectual reviews
- at the **source level**, to replace testing
  
  provided the correspondence with the binary is also certified

⇒ formal methods can improve cost-effectiveness!

Caveat: soundness is required by DO
Verification in practice: Formal verification at Airbus

Program proofs: deductive methods
- **functional** properties of *small sequential* C codes
- replace unit testing
- not fully automatic
- **Caveat / Frama-C** tool (CEA)

Sound static analysis:
- **fully** automated on *large* applications, *non functional* properties
- worst-case execution time and stack usage, on binary **aiT, StackAnalyzer** (AbsInt)
- absence of run-time error, on *sequential* C code **Astrée** analyzer (AbsInt)

Certified compilation:
- allows *source-level* analysis to **certify** sequential binary code
- **CompCert** C compiler, certified in **Coq** (INRIA)
Overview of abstract interpretation
Overview of abstract interpretation

Abstract interpretation

Patrick Cousot

General theory of the approximation and comparison of program semantics:

- unifies existing semantics (proposed independently)
- guides the design of static analyses that are correct by construction

---

Concrete semantics

\( (S_0) \)
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
Concrete semantics

\( (S_0) \)

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

\( S_i \in D = \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z}) \)

\( S_0 = \{ (i, x) | i, x \in \mathbb{Z} \} \quad = \top \)

\( S_1 = \{ (i, x) \in S_0 | x \in [0, 1000] \} \quad = F_1(S_0) \)

\( S_2 = \{ (0, x) | \exists i, (i, x) \in S_1 \} \quad = F_2(S_1) \)

\( S_3 = S_2 \cup S_5 \)

\( S_4 = \{ (i, x) \in S_3 | i < x \} \quad = F_4(S_3) \)

\( S_5 = \{ (i + 2, x) | (i, x) \in S_4 \} \quad = F_5(S_4) \)

\( S_6 = \{ (i, x) \in S_3 | i \geq x \} \quad = F_6(S_3) \)

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

- strongest program properties (inductive invariants)
- smallest solution of a system of equations, on sets
- well-defined solution, but not computable in general
Abstracting

**Principle:** be tractable by reasoning at an abstract level
Principle: be tractable by reasoning at an abstract level

Concrete executions: \{ (0, 3), (5.5, 0), (12, 7), \ldots \} (not computable)
**Principle:** be tractable by reasoning at an abstract level

Concrete executions: \{(0, 3), (5.5, 0), (12, 7), \ldots\} (not computable)

Box domain: \[X \in [0, 12] \land Y \in [0, 8]\] (linear cost)
Abstracting

**Principle:** be tractable by reasoning at an abstract level

- **Concrete executions:** \( \{(0,3), (5.5,0), (12,7), \ldots \} \) (not computable)
- **Box domain:** \( X \in [0,12] \land Y \in [0,8] \) (linear cost)
- **Polyhedra domain:** \( 6X + 11Y \geq 33 \land \cdots \) (exponential cost)

Many abstractions: trade-off cost vs. precision and expressiveness
From concrete to abstract semantics

\((S_0)\)
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 concrete semantics

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

- strongest program properties (inductive invariants)
- smallest solution of a system of equations
- not computable in general
From concrete to abstract semantics

\((S_0)\)
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 semantics

Abstract semantics \(S_i^\# \in D^\#\):

- \(D^\#\) is a subset of properties of interest
  - semantic choice + a machine representation
- \(F^\# : D^\# \rightarrow D^\#\) over-approximates the effect of \(F : D \rightarrow D\) in \(D^\#\)
  - with effective algorithms
Abstract interpretation

Define an interpretation of atomic statements in the abstract, and compose them to analyze the program

- by propagation along the edges of the control-flow graph (data-flow)
- or by induction on the syntax of programs (interpretation)

**Example in the polyhedra domain**

**Assignment:**
- $X = X + 1$
- translation

**Join:**
- if \ldots then \ldots else \ldots fi
- convex hull

**Loops or CFG cycles:**
- iteration with widening
Goal: prove that a program $P$ satisfies its specification $S$

We collect the reachable states $P$ and compare to $S$

A polyhedral abstraction $A$ can prove the correctness
Overview of abstract interpretation

Soundness and false alarms

Goal: prove that a program $P$ satisfies its specification $S$

We collect the reachable states $P$ and compare to $S$

A polyhedral abstraction $A$ can prove the correctness

A box abstraction cannot prove the correctness

$\implies$ false alarm

(especially since the analysis may not output the tightest box / polyhedron!)
Overview of abstract interpretation

Soundness and false alarms

Goal: prove that a program $P$ satisfies its specification $S$

We collect the reachable states $P$ and compare to $S$

A polyhedral abstraction $A$ can prove the correctness

A box abstraction cannot prove the correctness

$\implies$ false alarm

(especially since the analysis may not output the tightest box / polyhedron!)

The analysis is sound: no false negative reported!
Example static analyzer: Astrée

Astrée: developed at ENS & INRIA by P. Cousot & al.
- analyzes embedded critical C software
  subset of C, no memory allocation, no recursivity → simpler semantics
- checks for run-time errors
  arithmetic overflows, array overflows, divisions by 0, pointer errors, etc. → non-functional
- specialized for control / command software
  with zero false alarm goal
  application domain specific abstractions

Airbus A380

2001–2004: academic success
proof of absence of RTE
on flight command

2009: industrialization
Overview of abstract interpretation

Example static analyzer: **Infer.AI** at Facebook

**Infer:**  [http://fbinfer.com/](http://fbinfer.com/)

- developed at Facebook (team formerly at Monoidics)
- **Infer.AI** is an analysis framework based on abstract interpretation
- open-source since 2015
- analyzes Java, C, C++, and Objective-C
- checks ThreadSafety (Java), Initialisation Order (C++), etc.
- modular, bottom-up interprocedural analysis
- targets the analysis of **merge requests** (small bits at a time)
- favors speed over soundness
  - pragmatic choices, based on “what engineers want”
  - no requirements for certification, unlike the avionics industry
- used in production
Course organisation
Course organisation

Course plan

- **foundation** of abstract interpretation (2 courses)
  - fixpoint program semantics
  - order and approximation theory
  - hierarchy of semantics

- **bricks** of static analyzers (5 courses)
  - numeric abstract domains
  - pointer and memory shape abstract domains
  - partitioning domains
  - domain combiners (reduced products, partitioning)

- domain-specific **static analyses** (9 courses)
  - analysis of control-command embedded programs
  - analysis of concurrent programs
  - analysis of program transformation
  - analysis of distributed systems
  - analysis of mobile systems
  - analysis of biological systems
Teaching team

Cezara Drăgoi

Jérôme Feret

Antoine Miné

Xavier Rival
Syllabus and exams


Visit regularly for:

- latest information on course dates
- course material
- course assignments
- M2 internship proposals, updated regularly

Exams:

- 50%: written mid-term exam
- 50%: oral final exam
  (read a scientific article, present it, answer questions)
Course organisation

Course material

Links available on the web-page:

- main material: slides
- course notes
  cover mainly foundations and numeric abstract domains

Course assignments

On the web page, *highly recommended* homework

- **exercises**: prove a theorem, solve a former exam, etc.
- **reading assignments**: an article related to the course
- **experiments**: use a tool

**Principle: self-evaluation**

Not evaluated by the teacher, no credit.
The solution to the exercises is also given.

**Additional material:**

- previous exams, with correction
- course bibliography in the slides (reading not mandatory)
- optional programming project (not evaluated)