Introduction

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

Antoine Miné

Year 2019–2020

Course 00

11 September 2019
Formal Verification: 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.
A classic example: Ariane 5, Flight 501

**Cause:** software error\(^5\)

- **arithmetic overflow** in unprotected data conversion from 64-bit float to 16-bit integer types\(^6\)

\[
P_{M\text{-DERIVE}(T_{ALG.E\_BH})} := \text{UC\_16S\_EN\_16NS (TDB.T\_ENTIER\_16S} \\
(1.0/C_{M\_LSB\_BH}) * G_{M\_INFO\_DERIVE(T_{ALG.E\_BH}))};
\]

- 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

A “simple” error…

---

\(^{5}\) J.-L. Lions et al., Ariane 501 Inquiry Board report.

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

This case triggered the first large scale static code analysis
(PolySpace Verifier, using abstract interpretation)
**Undecidability:** correctness properties are undecidable!

cannot build a program that automatically and precisely separates all correct programs from all incorrect ones

Compromises:
lose automation, completeness, soundness, or generality

- **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)
- **Static analysis** (next slide)
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)
Verification in practice: The example of avionics software

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
  - provided the correspondence with the binary is also certified

⇒ **formal methods can improve cost-effectiveness!**

Caveat: **soundness** is required by DO
Formal Verification: Motivation

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
- guides the design of static analyses that are **correct by construction**

---

Concrete collecting 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 collecting semantics

\( \mathcal{S}_0 \)

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

\( \mathcal{S}_1 \)

\( I := 0; \)

\( \mathcal{S}_2 \)

while \((\mathcal{S}_3)\) \( I < X \) do

\( \mathcal{S}_4 \)

\( I := I + 2; \)

\( \mathcal{S}_5 \)

program semantics

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

- strongest program properties (inductive invariants)
- set of reachable environments, at each program point
- smallest solution of a system of equations
- well-defined solution, but not computable in general
Abstracting

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

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

concrete executions: \( \{(0, 3), (5.5, 0), (12, 7), \ldots \} \) (not computable)
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: \(6X + 11Y \geq 33\) (exponential 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

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

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

\( (S_2) \)
\[ \text{while } (S_3) I < X \text{ do} \]
\[ I := I + 2; \]

\( (S_5) \)

\( (S_6) \)

program concrete semantics

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

- \( [X \in [0, 1000]] \), \([I \leftarrow 0]\), etc. are transfer functions
- strongest program properties
- set of reachable environments, at each program point
- not computable in general
Overview of abstract interpretation

From concrete to abstract semantics

\begin{align*}
(S_0) & \quad \text{assume } X \text{ in } [0,1000]; \\
(S_1) & \quad I := 0; \\
(S_2) & \quad \text{while } (S_3) \ I < X \text{ do} \\
\quad & \quad (S_4) \quad I := I + 2; \\
(S_5) & \quad (S_6) \quad \text{abstract semantics}
\end{align*}

\begin{align*}
S_i & \in \mathcal{D}^\# \\
S_0^\# & = \top^\# \\
S_1^\# & = [[ X \in [0,1000] ]]^\#(S_0^\#) \\
S_2^\# & = [ I \leftarrow 0 ]^\#(S_1^\#) \\
S_3^\# & = S_2^\# \cup S_5^\# \\
S_4^\# & = [ I < X ]^\#(S_3^\#) \\
S_5^\# & = [ I \leftarrow I + 2 ]^\#(S_4^\#) \\
S_6^\# & = [ I \geq X ]^\#(S_3^\#)
\end{align*}

\textbf{Abstract semantics } $S_i^\# \in \mathcal{D}^\#$:

- $\mathcal{D}^\#$ is a subset of properties of interest
  - semantic choice + a machine representation
- $F^\# : \mathcal{D}^\# \to \mathcal{D}^\#$ over-approximates the effect of $F : \mathcal{D} \to \mathcal{D}$ in $\mathcal{D}^\#$
  - with effective algorithms
Abstract operator examples

In the polyhedra domain:

- **Abstract assignment**
  \[
  \llbracket X \leftarrow X + 1 \rrbracket^\# \\
  \text{translation (exact)}
  \]

- **Abstract union**
  \[
  \bigcup^\# \\
  \text{convex hull (approximate)}
  \]

- **Solving the equation system**
  by iteration
  using **extrapolation** to terminate
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
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

$A \not\subseteq S$ → false alarm

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

- 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
Teaching team

Cezara Drăgoi

Antoine Miné

Jérôme Feret

Xavier Rival
Syllabus and exams

Visit regularly for:

- latest information on course dates
- course material
- course assignments
- internship proposals

**Exams:**

- 50%: written mid-term exam (3h)
- 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 based on:


- recommended reading on theory and applications:

Course organisation

Course assignments (self-evaluated)

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

Also:

- previous exams, with correction
- example programming project
  (abstract interpreter for a toy language in OCaml)

**Principle**: self-evaluation

No credit.
Not evaluated by the teacher.
Foundations of abstract interpretation: (courses 1 & 2)

- Mathematical background: order theory and fixpoints
- Formalization of abstraction, soundness
- Program semantics and program properties
- Hierarchy of collecting semantics

\[ a \leq \alpha(c) \]
Basic bricks of abstraction: numerical domains (courses 3, 4 & 15)

simple domains

- **Intervals**
  \[ x \in [a, b] \]

- **Congruences**
  \[ x \in a\mathbb{Z} + b \]

relational domains

- **Octagons**
  \[ \pm x \pm y \leq c \]

- **Polyhedra**
  \[ \sum_i \alpha_i x_i \leq \beta \]

specific domains

- **Ellipsoids**
- **digital filters**

- **Exponentials**
  **rounding errors**
Basic bricks of abstraction: memory abstractions (courses 7 & 11)

- beyond numeric: reason on arrays, lists, trees, graphs, …
- challenges: variety of structures, destructive updates
- logical tools:
  - separation logics (a logic tailored for describing memory)
  - parametric three valued logics (representing arbitrary graphs)
- abstract domains based on these logics
Basic bricks of abstraction: partitioning abstractions (course 10)

- most abstract domains are not distributive
  $\implies$ reasoning over disjunctions loses precision
- first solution: add disjunctions to any abstract domain
  $\implies$ expressive but costly
- second solution: partitioning
  conjunctions of implications as logical predicates
  (partitioning may be based on many semantic criteria)
Course plan (5/8)

**Analyses:** analysis of concurrent data-structures (courses 8 & 9)

- abstract domains to reason about relations between data structures
- thread-modular abstractions
- program logic combing rely-guarantee and separation logic
- concurrent data-structure verification (reduction to state reachability provable by the abstract domains)
Course organisation

Course plan (6/8)

**Analyses:** analysis of mobile systems *(courses 12 & 13)*

- dynamic creation of components and links
- analyze the links between components
  - distinguish between recursive components
  - abstractions as *sets of words*
- bound the number of components
  using numeric relations

![Diagram](image)
Analyses: abstractions of signaling pathways (courses 5 & 6)

[Eikuch, 2007]
Analyses: abstractions of signaling pathways (courses 5 & 6)
abstractions offer different perspectives on models

Concrete semantics

Causal traces

Information flow

Exact projection of the ODE semantics
Analyses: static analysis for security (course 16)

- challenge: security properties are diverse from information leakage to unwanted execution of malicious code and more complex than safety and liveness
- the framework of hyperproperties can express security
- apply abstract interpretation to reason over non-interference
Internship proposals

Possibility of Master 2 internships at ENS or Sorbonne Université.

Example topics:

- Static analysis of smart contracts
- Semantic input data usage analysis
- Algorithmic fairness analysis of neural networks
- Counter-example generation through backward under-approximations
- Static analysis for lock-free data structures
- Static analysis for consensus algorithms
- ...

Formal proposals will be available on the course page also: discuss with your teachers for tailor-made subjects.