Introduction

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

Antoine Miné

Year 2021–2022

Course 0

20 September 2021
Formal Verification: Motivation
A classic example: Ariane 5, Flight 501

Maiden flight of the Ariane 5 Launcher, 4 June 1996.
Cost of failure estimated at more than 370 000 000 US$\(^1\)

Cause of Ariane 5 failure

**Cause:** software error

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

\[
P_M_{\text{DERIVE}}(T_{\text{ALG.E.BH}}) := \\
\text{UC}_{16S\_\text{EN.16NS}}(\text{TDB.T\_ENTIER.16S} \\
(1.0/C_M_{\text{LSB.BH}}) \times G_M_{\text{INFO.DERIVE}}(T_{\text{ALG.E.BH}})) ;
\]

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

A “simple” error...
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

This case triggered the first large scale static code analysis
(PolySpace Verifier, using abstract interpretation)
Verification: compromises

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)
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 standards
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)
Another example bug: Heartbleed

Vulnerability in OpenSSL cryptographic library all versions from 2012 to 2014
OpenSSL is used 66% of WEB servers for https (also: email encryption, VPN, etc.)

Cause: buffer overflow in “heartbeat” protocol.
Consequence:
- leak of private information, such as private keys
- no way to actually know what has been extracted
  \[\Rightarrow\] need to renew all keys after correcting the bug!
- very high economic cost!

\[4\url{http://heartbleed.com}\]
Improving software quality

Study from Consortium for Information & Software Quality:\textsuperscript{5}

- $607 \text{ billions spent finding and fixing bugs}$
- $1.56 \text{ trillion cost for software failure}$
- just for 2020 in the US!

$\Rightarrow$ even non-critical domains could use \textit{formal methods}!

\textsuperscript{5}Herb Krasner. The cost of poor software quality in the US: A 2020 report. 
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

\[(S_0)\] 
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 \(S_i \in \mathcal{D} = \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z})\):

- 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
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)

Many abstractions: trade-off cost vs. precision and expressiveness
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 \([0,1000]\);
\(S_i \in \mathcal{D} \overset{\text{def}}{=} \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z})\)

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

\((S_2)\)
while \((S_3)\) \(I < X\) do
\((S_4)\)
\(I := I + 2;\)

\((S_5)\)
\(I \leftarrow I + 2\)

\((S_6)\)
\(I \geq X\)

Concrete semantics \(S_i \in \mathcal{D} = \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z})\):
- \([X \in [0,1000]]\), \([I \leftarrow 0]\), etc. are transfer functions
- strongest program properties
- set of reachable environments, at each program point
- not computable in general

Program concrete semantics.
From concrete to abstract semantics

\[(S_0)\]
\[\text{assume } X \text{ in } [0, 1000];\]
\[(S_1)\]
\[I := 0;\]
\[(S_2)\]
\[\text{while } (S_3) I < X \text{ do}\]
\[(S_4)\]
\[I := I + 2;\]
\[(S_5)\]

program

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

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
Overview of abstract interpretation

Abstract operator examples

In the polyhedra domain:

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

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

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

$\Rightarrow$ 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!
eBPF:

- a virtual machine inside the Linux kernel
- can run arbitrary code in kernel mode
  - very low-level, can perform arbitrary pointer arithmetic
- run sandboxed to protect against bugs and attacks

In theory:

- a static analysis checks bytecode safety before execution
- includes an interval analysis for pointers
Bound computation for bit-shift $\gg$:

```c
case BPF_RSH:
    if (min_val < 0 || dst_reg->min_value < 0)
        dst_reg->min_value = BPF_REGISTER_MIN_RANGE;
    else
        dst_reg->min_value = (u64)(dst_reg->min_value) >> min_val;
    if (dst_reg->max_value != BPF_REGISTER_MAX_RANGE)
        dst_reg->max_value >>= max_val;
    break;
```

Due to large amount of bugs in the static analysis, a dynamic analysis has been added... which exploits results from by the static analysis...
Example tools
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
Infer.AI

**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
Frama-C:  https://frama-c.com/

- developed at CEA
- open-source
- analyzes C
- combines abstract interpretation and deductive methods
- has a specification language (ACSL) for functional verification
- used in industrial applications
Research project: **MOPSA**

Modular Open Platform For Static Analysis developed at Sorbonne University: [https://mopsa.lip6.fr/](https://mopsa.lip6.fr/)

An abstract interpreter prototype tool for research and education

- extendable to new properties and new languages
- help developing, reusing, combining abstractions
- open-source: [https://gitlab.com/mopsa/mopsa-analyzer](https://gitlab.com/mopsa/mopsa-analyzer)

Currently available: (but not fully scalable)

- C analysis for run-time error detection
- Python analysis

On-going research:

- patch and portability analysis for C
- analyze programs mixing C and Python
- analysis of smart-contracts
- internship possible!
Course organisation
Teaching team

Caterina Urban

Jérôme Feret

Antoine Miné

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:

  A. Miné. *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation.*

- **recommended reading on theory and applications:**

  J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival. *Static analysis and verification of aerospace software by abstract interpretation.*
Course organisation

Course assignments (self-evaluation)

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 corrected by teachers.
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
Bricks of abstraction: numerical domains

- 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
Bricks of abstraction: memory abstractions

- 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
Bricks of abstraction: partitioning abstractions

- most abstract domains are not distributive
  \[\Rightarrow\] reasoning over disjunctions loses precision

- first solution: add disjunctions to any abstract domain
  \[\Rightarrow\] expressive but costly

- second solution: partitioning
  conjunctions of implications as logical predicates
  (partitioning may be based on many semantic criteria)
**analyses:** abstract interpretation for liveness properties

- beyond safety (e.g., absence of errors)
  we prove that **programs (eventually) do something good**

- abstract domains to reason about **program termination**
  inference of **ranking functions**

- generalization to **other liveness properties**
  (e.g., expressed in **CTL**)

---

**course organisation**

**course plan (5/8)**

**introduction**

Antoine Miné

Course 0 | Introduction | Antoine Miné | p. 35 / 39
Analyses: static analysis of neural networks

- verification of local robustness against adversarial examples

- fairness certification
  (special case of global robustness verification)

- verification of functional properties
Analyses: analysis of mobile systems

- 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
Analyses: static analysis for security

- 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:

- Automatic inference of input data assumptions
- Fairness certification of machine-learned software
- Static analysis of functional languages
- Inferring counter-examples through static analysis
- Static analysis of medical data processing software
- Static analysis for lock-free data structures
- Static analysis for consensus algorithms
- ...

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