

# Introduction

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

Antoine Miné

Year 2024–2025

Course 0  
23 September 2024



# Formal Verification: Motivation

---

# Historic 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\$<sup>1</sup>

---

<sup>1</sup> M. Dowson. "The Ariane 5 Software Failure". Software Engineering Notes 22 (2): 84, March 1997.

# Cause of Ariane 5 failure

**Cause:** software error<sup>2</sup>

- arithmetic overflow in unprotected data conversion from 64-bit float to 16-bit integer types<sup>3</sup>

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

A “simple” error. . .

---

<sup>2</sup> J.-L. Lions et al., Ariane 501 Inquiry Board report.

<sup>3</sup> 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, 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! (Rice's theorem)  
no program can automatically and precisely separates all correct programs from all incorrect ones

Compromises:    **lose** automation, or completeness, or soundness, or generality

- **Test, symbolic execution:** complete and automatic, but unsound
- **Theorem proving**
  - proof essentially manual, but checked automatically
  - powerful, but very steep learning curve and large effort required
- **Deductive methods**
  - automated proofs for some logic fragments (SAT, SMT)
  - still requires some 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

## source

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    if (t[i]) break;
  }
  return t[i];
}
```

⇒

## analysis result

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    //  $0 \leq i < n$ 
    if (t[i]) break;
  }
  //  $0 \leq i \leq n \vee n < 0$ 
  return t[i];
}
```

✓  
✗

- work directly on the **source code**
- **infer** properties on **program executions**
- **automatically** (cost effective)
- by constructing dynamically a **semantic abstraction** of the program
- to deduce program **correctness**, or raise **alarms** if it cannot  
implicit specification: absence of RTE; or (simple) user-defined properties: contracts
- with **approximations** (incomplete: efficient, but possible false alarms)
- **soundly** (no false positive)



# Verification in practice: Avionics software

Critical avionics software is subject to **certification**:

- **70%** of the development cost (in 2015)
- regulated by **international standards** (DO-178)
- 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 that 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 by 66% of WEB servers for <https>  
(also: email encryption, VPN, etc.)

Cause:    **buffer overflow** in “heartbeat” protocol

Consequence:<sup>4</sup>

- leak of private information, e.g., private keys
- no way to actually know which keys have been compromised  
⇒ need to renew all keys after correcting the bug!
- very high economic cost!

---

<sup>4</sup> <http://heartbleed.com>

# The need to improve general software quality

Recent study from [Consortium for Information & Software Quality](#):<sup>5</sup>

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

⇒ non-critical domains could benefit from **formal methods**!

## Challenges:

- keep up with scalability
- more complex languages and programming models (C++, JavaScript, Python, ...)
- go beyond absence of run-time errors and towards functional properties
- increase usability (error classification, explanation, ...)
- while still being sound!

---

<sup>5</sup> Herb Krasner. The cost of poor software quality in the US: A 2020 report. <https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf>, 2021. Accessed: 2021-08.

# Overview of abstract interpretation

---

# Abstract interpretation



Patrick Cousot



Radhia Cousot



P. Cousot's PhD<sup>6</sup>

General theory of the approximation and comparison of program semantics:

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

<sup>6</sup>P. Cousot. "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes." Thèse Ès Sciences Mathématiques, 1978.

# 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;`

$(\mathcal{S}_5)$

$(\mathcal{S}_6)$

`program`

# Concrete collecting semantics

$(S_0)$	
assume $X$ in $[0,1000]$ ;	$S_i \in \mathcal{D} = \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z})$
$(S_1)$	$S_0 = \{ (i, x) \mid i, x \in \mathbb{Z} \} = \top$
$I := 0;$	$S_1 = \{ (i, x) \in S_0 \mid x \in [0, 1000] \} = F_1(S_0)$
$(S_2)$	$S_2 = \{ (0, x) \mid \exists i, (i, x) \in S_1 \} = F_2(S_1)$
while $(S_3)$ $I < X$ do	$S_3 = S_2 \cup S_5$
$(S_4)$	$S_4 = \{ (i, x) \in S_3 \mid i < x \} = F_4(S_3)$
$I := I + 2;$	$S_5 = \{ (i + 2, x) \mid (i, x) \in S_4 \} = F_5(S_4)$
$(S_5)$	$S_6 = \{ (i, x) \in S_3 \mid i \geq x \} = F_6(S_3)$
$(S_6)$	
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**

# Abtracting

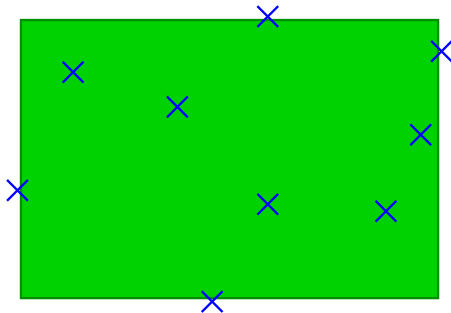
Principle: be tractable by reasoning at an **abstract level**



concrete executions :  $\{(0, 3), (5.5, 0), (12, 7), \dots\}$  (not computable)

# Abtracting

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

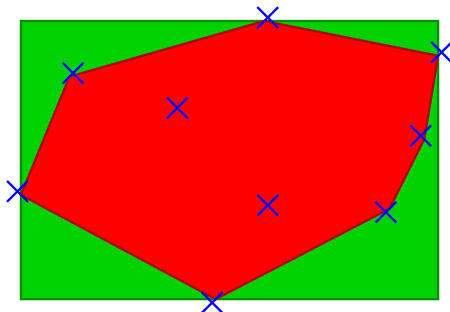


concrete executions :  $\{(0, 3), (5.5, 0), (12, 7), \dots\}$  (not computable)

box domain :  $X \in [0, 12] \wedge Y \in [0, 8]$  (linear cost)

# Abtracting

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



concrete executions :  $\{(0, 3), (5.5, 0), (12, 7), \dots\}$  (not computable)

box domain :  $X \in [0, 12] \wedge Y \in [0, 8]$  (linear cost)

polyhedra domain :  $6X + 11Y \geq 33 \wedge \dots$  (exponential cost)

many abstractions: trade-off cost vs. precision and expressiveness

# From concrete to abstract semantics

$(S_0)$

assume  $X$  in  $[0, 1000]$ ;  $S_i \in \mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\{I, X\} \rightarrow \mathbb{Z})$

$(S_1)$

$I := 0;$   $S_0 = \{(i, x) \mid i, x \in \mathbb{Z}\}$   
 $S_1 = \llbracket X \in [0, 1000] \rrbracket (S_0)$

$(S_2)$

while  $(S_3)$   $I < X$  do  $S_2 = \llbracket I \leftarrow 0 \rrbracket (S_1)$

$(S_4)$

$I := I + 2;$   $S_3 = S_2 \cup S_5$   
 $S_4 = \llbracket I < X \rrbracket (S_3)$

$(S_5)$

$S_5 = \llbracket I \leftarrow I + 2 \rrbracket (S_4)$   
 $S_6 = \llbracket I \geq X \rrbracket (S_3)$

$(S_6)$

program

concrete semantics

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

- $\llbracket X \in [0, 1000] \rrbracket$ ,  $\llbracket I \leftarrow 0 \rrbracket$ , etc. are transfer functions
- strongest program properties
- set of reachable environments, at each program point
- 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

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

abstract semantics

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

- $\mathcal{D}^\#$  is a subset of properties of interest  
semantic choice + machine representation
- $F^\# : \mathcal{D}^\# \rightarrow \mathcal{D}^\#$  over-approximates the effect of  $F : \mathcal{D} \rightarrow \mathcal{D}$  in  $\mathcal{D}^\#$   
abstract operators proved sound + effective algorithms

# Abstract operator examples

In the polyhedra domain:

- Abstract assignment

$\llbracket X \leftarrow X + 1 \rrbracket^\#$

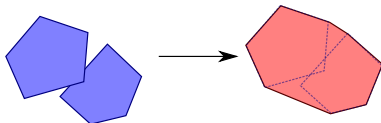
translation (exact)



- Abstract union

$\cup^\#$

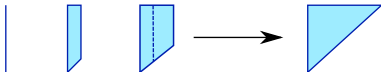
convex hull (approximate)



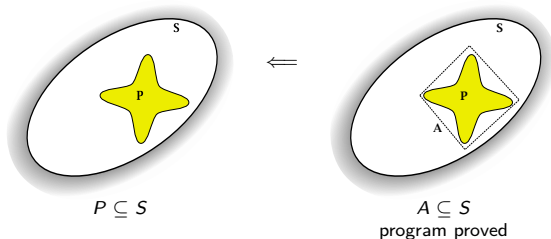
- Solving the equation system

by iteration

using extrapolation to terminate



# Soundness and false alarms



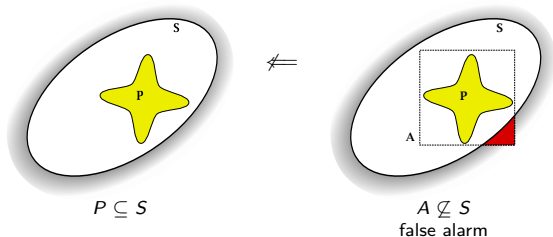
Goal: prove that a program  $P$  satisfies its specification  $S$

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  $S$

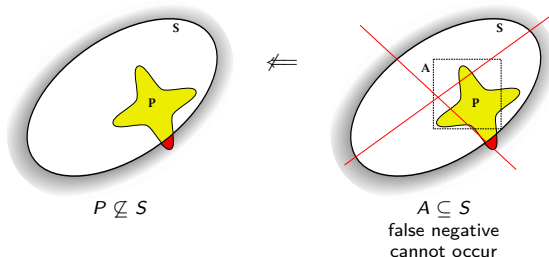
A **polyhedral abstraction**  $A$  can prove the correctness

A **box abstraction** cannot prove the correctness

$\implies$  false alarm

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

# 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!

# The need for formal justification : eBPF example

## eBPF:

- a virtual machine in the Linux kernel
- can **run arbitrary code in kernel mode**
- very low-level, can perform arbitrary pointer arithmetic (flat memory model)
- a **static analysis** checks bytecode safety before execution (interval analysis)

(Incorrect) bound computation for bit-shifts >>:<sup>7</sup>

```
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;
```

## Lesson

Use abstract interpretation to make analyses sound by construction!

<sup>7</sup> [www.zerodayinitiative.com/blog/2021/1/18/zdi-20-1440-an-incorrect-calculation-bug-in-the-linux-kernel-ebpf-verifier](https://www.zerodayinitiative.com/blog/2021/1/18/zdi-20-1440-an-incorrect-calculation-bug-in-the-linux-kernel-ebpf-verifier)

# Example tools

---

# Astrée

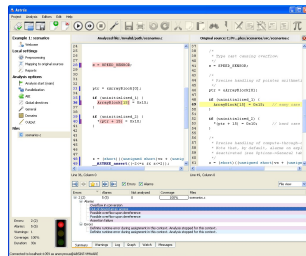
**Astrée:** developed at ENS & INRIA by P. Cousot & al.

- 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

**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

# MOPSA

## Modular Open Platform for Static Analysis

- **research tool** developed at Sorbonne Université: <https://mopsa.lip6.fr/>
- **extendable** to new properties and new languages
- help developing, reusing, combining abstractions
- **open-source**: <https://gitlab.com/mopsa/mopsa-analyzer>

## Analyses

- C analysis for run-time error detection (Coreutils, Juliet)
- Python analysis (supports a large subset of Python 3, and a small subset of its library)
- OCaml analysis (work in progress)
- patch and portability analysis
- taint and value analyses (security, exploitability)

## Possible research topics: supported by MOPSA

- multi-language analyses
- dependency and impact analyses
- functional properties, user-specified properties
- function-modular analyses



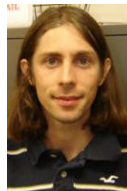
# Course organisation

---

# Teaching team



Caterina Urban



Jérôme Feret



Antoine Miné



Xavier Rival

# Syllabus and exams

<https://www-apr.lip6.fr/~mine/enseignement/mpri/2024-2025>

Visit **regularly** for:

- latest information on course dates and modalities and, possibly, **last-minute changes**
- course material (slides)
- optional course assignments and reading
- internship proposals

## Exams:

- 50%: **written** mid-term exam (3h)
- 50%: **oral** final exam  
(read a scientific article, present it, answer questions)

# Course material

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\*. In Foundations and Trends in Programming Languages, 4\(3–4\), 120–372. Now Publishers.](#)

- 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\*. In Foundations and Trends in Programming Languages, 2\(2–3\), 71–190, 2015. Now Publishers.](#)

# Course assignments (self-evaluation)

On the web page, **recommended homework**

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

Also:

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

Principle: self-evaluation

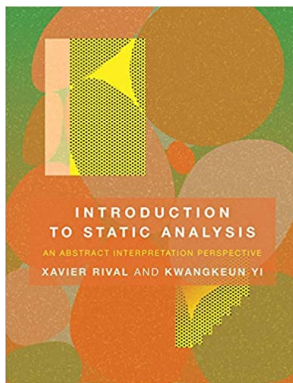
- no credit
- not corrected by the teachers

# Recent books!

## PRINCIPLES OF ABSTRACT INTERPRETATION



PATRICK COUSOT

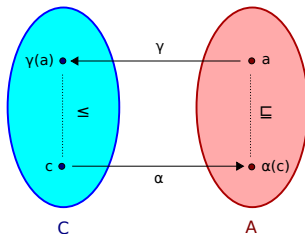


- 1 P. Cousot. Principles of Abstract Interpretation. 832 pages. The MIT Press. Sept. 2021.
- 2 X. Rival and K. Yi. Introduction to Static Analysis: An Abstract Interpretation Perspective. 320 pages. The MIT Press. Feb, 2020.

# Course plan (1/8)

## Foundations of abstract interpretation: (courses 1 & 2)

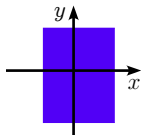
- mathematical background: **order theory** and **fixpoints**
- formalization of **abstraction**, soundness
- program **semantics** and program **properties**
- **hierarchy** of collecting semantics



# Course plan (2/8)

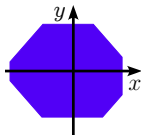
## Bricks of abstraction: numerical domains

simple domains



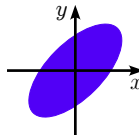
Intervals  
 $x \in [a, b]$

relational domains

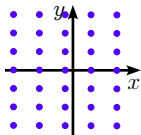


Octagons  
 $\pm x \pm y \leq c$

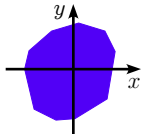
specific domains



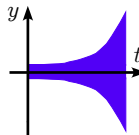
Ellipsoids  
digital filters



Congruences  
 $x \in a\mathbb{Z} + b$



Polyhedra  
 $\sum_i \alpha_i x_i \leq \beta$



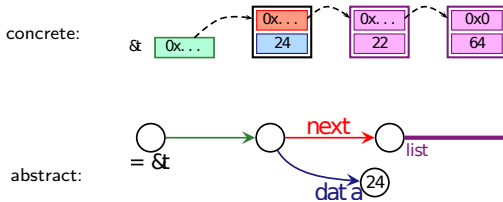
Exponentials  
rounding errors



# Course plan (3/8)

## Bricks of abstraction: memory abstractions

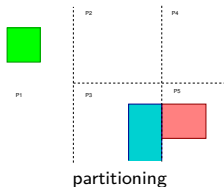
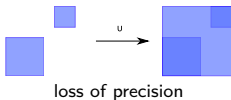
- beyond numeric: reason on **arrays**, **lists**, **trees**, **graphs**, ...
- challenges: variety of structures, destructive updates
- logical tools:
  - **separation logics** (a logic tailored for describing memory)
  - **parametric three valued logics** (representing arbitrary graphs)
- **abstract domains** based on these logics



# Course plan (4/8)

## Bricks of abstraction: partitioning abstractions

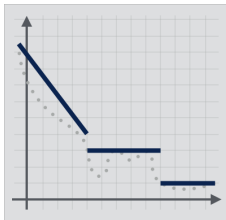
- most abstract domains are **not distributive**  
 $\implies$  reasoning over disjunctions **loses precision**
- first solution: **add disjunctions** to any abstract domain  
 $\implies$  expressive but costly
- second solution: **partitioning**  
conjunctions of implications as logical predicates 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



# Course plan (5/8)

## Analyses: abstract interpretation for liveness properties

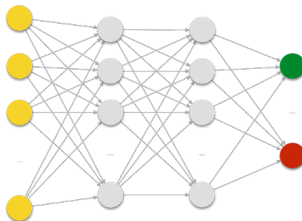
- beyond safety (e.g., absence of errors)  
we prove that **programs (eventually) do something good**
- abstract domains to reason about **program termination**  
inference of **ranking functions**



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

# Course plan (6/8)

## Analyses: static analysis of neural networks

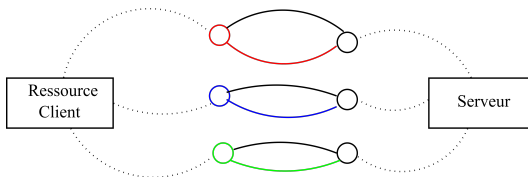


- verification of **local robustness** against **adversarial examples**
- **fairness** certification  
(special case of **global robustness** verification)
- verification of **functional properties**

# Course plan (7/8)

## 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



# Course plan (8/8)

## 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

Master 2 internship proposals at ENS, Sorbonne Université or INRIA Lille, possibly followed by a PhD

## Example topics:

- Static analysis for multi-language programs
- Static analysis for user-specified properties
- Static analysis of smoothness properties
- Determining the impact of a change using semantic dependencies
- Static analysis under a time budget
- Static analysis of the robustness of machine-learning software
- Abstract domain reductions between separation logic and value abstractions
- ...

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