Introduction
Semantics and applications to verification

Xavier Rival

École Normale Supérieure
Program of this first lecture

Introduction to the course:

1. a study of some examples of software errors
   - their consequences in real systems
   - we will look for an understanding of their root cause

2. a panel of the main verification methods
   with a fundamental limitation: decidability
   - many techniques allow to compute semantic properties
   - each comes with advantages and drawbacks

3. an introduction to the theory of ordered sets
   - order relations are pervasive in semantics and verification
   - fixpoints of operators are also very common
Outline

1 Introduction

2 Case studies
   - Patriot missile (anti-missile system), Dahran (1991)
   - General remarks

3 Approaches to verification

4 Orderings, lattices, fixpoints
Ariane 5 – Flight 501

- **Ariane 5:**
  - a satellite launcher
  - replacement of *Ariane 4*, a lot more powerful
  - first flight, June, 4th, 1996: failure!

- **Flight story:**
  - nominal take-off, normal flight for 36 seconds
  - $T + 36.7 \text{ s} $: angle of attack change, trajectory lost
  - $T + 39 \text{ s} $: disintegration of the launcher

- **Consequences:**
  - **loss of satellites** : more than $370\,000\,000$
  - **launcher** unusable for more than a year (delay!)
  - impact on reputation (*Ariane 4* was very reliable)

- **Full report available online:**
  - [http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf](http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf)

Jacques-Louis Lions, Gilles Kahn
Trajectory control system design overview

- **Sensors:** gyroscopes, inertial reference systems...
- **Calculators** (hardware + software):
  - “Inertial Reference System” (SRI):
    integrates data about the trajectory (read on sensors)
  - “On Board Computer” (OBC):
    computes the engine actuations that are required to follow the pre-determined theoretical trajectory
- **Actuators:** engines of the launcher follow orders from the OBC
- **Redundant systems** (failure tolerant system):
  - keep running even in the presence of one or several system failures
  - traditional solution in embedded systems: duplication of systems
    aircraft flight system: 2 or 3 hydraulic circuits
    launcher like Ariane 5: 2 SRI units (SRI 1 and SRI 2)
  - there is also a control monitor
The root cause: an unhandled arithmetic error

**Processor registers**

Each register has a size of 16, 32, 64 bits:

- **64-bits floating point**: values in range $[-3.6 \cdot 10^{308}, 3.6 \cdot 10^{308}]$
- **16-bits signed integers**: values in range $[-32768, 32767]$
- upon **copy of data**: conversions are performed such as rounding
- when the values are **too large**:
  - **interruption**: run error handling code if any, otherwise crash
  - or **unexpected behavior**: modulo arithmetic or other

**Ariane 5:**

- the **SRI** hardware runs in **interruption mode**
- it has **no error handling code** for arithmetic interruptions
- the **root cause** is an unhandled arithmetic conversion overflow
A **not so trivial** sequence of events:

1. a **conversion from 64-bits float to 16-bits signed int overflows**
2. an **interruption** is raised
3. due to the lack of error handling code, the SRI **crashes**
4. the crash causes an **error return** (negative integer value) value be **sent to the OBC** (On-Board Computer)
5. the OBC interprets this illegal value as **flight data**
6. this causes the computation of an **absurd trajectory**
7. hence the **loss of control** of the launcher
Addressing the software error

Several solutions would have prevented this misshappening:

1. **Deactivate interruptions on overflows:**
   - then, an overflow may happen, and cause wrong values be manipulated in the SRI
   - but, these wrong values will not cause the computation to stop!
   - and most likely, the flight will not be impacted too much

2. **Fix the SRI code**, so that no overflow can happen:
   - all conversions must be **guarded against overflows**:
     ```
     double x = ...;
     short k = ...;
     if(−32768. ≤ x && x ≤ 32767.)  i = (short)x;
     else  i = ...;
     ```
   - this may be costly (many tests), but redundant tests can be removed

3. **Handle** conversion errors (not trivial):
   - the handling code should **identify the problem** and **fix it** at run-time
   - the OBC should **identify illegal input values**
A crash due to a useless task

Piece of code that generated the error:
- part of a gyroscope re-calibration process
- very useful to quickly restart the launch process after a short delay
- can only be done before lift-off...
- ... but not after!

Re-calibration task shut down:
- normally planned 50 seconds after lift-off...
- no chance of a need for such a re-calibration after $T_0 + 3$ seconds
- the crash occurred at 36 seconds
A crash due to legacy software

**Software history:**
- already used in Ariane 4 (previous launcher, before Ariane 5)
- the software was tested and ran in real conditions many times yet never failed...
- but Ariane 4 was a much less powerful launcher

**Software optimization:**
- many conversions were *initially protected by a safety guard*
- but these tests were considered *expensive*
  (a test and a branching take processor cycles, interact with the pipeline...)
- thus, conversions were ultimately *removed* for the sake of performance

Yet, Ariane 5 violates the assumptions that were valid with Ariane 4
- higher values of *horizontal bias* were generated
- those were never seen in Ariane 4, hence the failure
A crash not prevented by redundant systems

**Principle of redundant systems:**
survive the failure of a component by the use of redundant systems

**System redundancy in Ariane 5:**
- one OBC unit
- **two SRI units**... yet **running the same software**

Obviously, physical redundancy does not address software issues

**System redundancy in Airbus FBW software:**
- two independent set of controls
- three computing units per set of controls
- each computing unit comprises **two computers**
  - **distinct softwares**
  - design and implementation is also performed in **distinct teams**
Ariane 501, a summary of the issues

A long series of design errors, all related to a lack of understanding of what the software does:

1. **Non-guarded** conversion raising an **interruption** due to **overflow**
2. **Removal of pre-existing guards**, too high confidence in the software
3. **Non revised assumptions** on the inputs when moving from Ariane 4 to Ariane 5
4. **Redundant systems** running the same software
5. **Useless task** not **shutdown** at the right time

**Current status**: such issues can be found by static analysis tools
Outline

1. Introduction

2. Case studies
   - Patriot missile (anti-missile system), Dahran (1991)
   - General remarks

3. Approaches to verification

4. Orderings, lattices, fixpoints
High-speed runway overshoot at landing

**Landing** at Warsaw airport, Lufthansa A320:

- **bad weather conditions:** rain, high side wind
- **wet runway**
- landing (300 km/h) followed by **aqua-planing**, and **delayed braking**
- **runway overrun** at 132 km/h
- **impact** against a hillside at about 100 km/h

**Consequences:**

- **2 fatalities, 56 injured** (among 70 passengers + crew)
- **aircraft completely destroyed** (impact + fire)

**Full report available online:**
http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html
Causes of the accident

1 Root cause:
   - **bad weather conditions** not well assessed by the crew
   - side wind **exceeding** aircraft certification specification
   - **wrong action from the crew:**
     a “Go Around” (missed landing, acceleration + climb) should have been done

2 Contributing factor: **delayed action of the brake system**

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Distance (meters) from runway threshold</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>770 m</td>
<td>main landing gear landed</td>
</tr>
<tr>
<td>$T_0 + 3$ s</td>
<td>1030 m</td>
<td>nose landing gear landed</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>brake command activated</strong></td>
</tr>
<tr>
<td>$T_0 + 12$ s</td>
<td>1680 m</td>
<td>spoilers activated</td>
</tr>
<tr>
<td>$T_0 + 14$ s</td>
<td>1800 m</td>
<td>thrust reversers activated</td>
</tr>
<tr>
<td>$T_0 + 31$ s</td>
<td>2700 m</td>
<td>end of runway</td>
</tr>
</tbody>
</table>
Protection of aircraft brake systems

- **Braking systems inhibition:** Prevent in-flight activation!
  - **spoilers:** increase in aerodynamic load (drag)
  - **thrust reversers:** could destroy the plane if activated in-flight!
    (ex: crash of a B 767-300 ER Lauda Air, 1991, 223 fatalities; thrust reversers in-flight activation, electronic circuit issue)

- **Braking software specification:**
  DO NOT activate spoilers and thrust reverse unless the following condition is met:
  - thrust lever should be set to **minimum** by the flight crew
  - **AND** either of the following conditions:
    - **weight** on the main gear should be at least **12 T**
      i.e., **6 T** for each side
    - **OR** wheels should be spinning, with a speed of at least 130 km/h

  \[\text{[Minimum Thrust]} \AND \text{([Weight] OR [Wheels spinning])}\]
Understanding the braking delay

- **Landing configuration:**

  - wind action
  - aquaplaning \(\Rightarrow\) no rotation
  - ground action (opp. weight)

- **Braking systems:** inhibited
  - thrust command properly set to minimum
  - no weight on the left landing gear due to the wind
  - no speed on wheels due to aquaplaning

  \[
  \text{[Minimum Thrust]} \land (\text{[Weight]} \lor \text{[Wheels spinning]})
  \]
Flight 2904, a summary of the issues

Main factor is human (landing in a weather the airplane is not certified for), but the specification of the software is a contributing factor:

- **Old condition** that failed to be satisfied:
  \[(P_{\text{left}} > 6T) \text{ AND } (P_{\text{right}} > 6T)\]

- **Fixed condition** (used in the new version of the software):
  \[(P_{\text{left}} + P_{\text{right}}) > 12T\]

- The fix can be understood only with knowledge of the environment
  - conditions which the airplane will be used in
  - behavior of the sensors
Outline

1 Introduction

2 Case studies
   - Patriot missile (anti-missile system), Dahran (1991)
   - General remarks

3 Approaches to verification

4 Orderings, lattices, fixpoints
The anti-missile “Patriot” system

- **Purpose**: destroy foe missiles before they reach the ground

- **Use in wars**:
  - **first Gulf war** (1991)
    protection of towns and military facilities in Israël and Saudi Arabia (against “Scud” missiles launched by Irak)
  - **success rate**:
    ★ around 50 % of the “Scud” missiles are successfully destroyed
    ★ almost all launched Patriot missiles destroy their target
    ★ failures are due to failure to launch a Patriot missile

- **Constraints on the system**:
  - **hit very quickly moving targets**:
    “Scud” missiles fly at around 1700 m/s ; travel about 1000 km in 10 minutes
  - **not to destroy a friendly target** (it happened at least twice!)
  - very high cost: about $1 000 000 per launch
System components

Detection / trajectory identification:

- detection using radar systems
- trajectory confirmation (to make sure a foe missile is tracked):
  1. trajectory identification using a sequence of points at various instants
  2. trajectory confirmation
     computation of a predictive window (from position and speed vector)
     + confirmation of the predicted trajectory
  3. identification of the target (friend / foe)

Guidance system:

- interception trajectory computation
- launch of a Missile, and control until it hits its target
  high precision required (both missiles travel at more than 1500 m/s)

Very short process: about ten minutes
Dahran failure (1991)

1. **Launch of a “Scud” missile**

2. **Detection** by the radars of the Patriot system but **failure to confirm the trajectory**
   - imprecision in the computation of the **clock** of the detection system
   - computation of a **wrong confirmation window**
   - the “Scud” cannot be found **in the predicted window**
   - failure to confirm the trajectory
   - the detection computer concludes it is a **false alert**

3. **The “Scud” missile hits its target:**
   - **28 fatalities** and around **100 people injured**
Fixed precision arithmetic

- **Fixed precision numbers** are of the form $\epsilon N 2^{-p}$ where:
  - $p$ is fixed
  - $\epsilon \in \{-1, 1\}$ is the **sign**
  - $N \in [-2^n, 2^n - 1] \mathbb{Z}$ is an **integer** ($n > p$)

- In **32 bits fixed precision**, with one sign bit, $n = 31$; thus we may let $p = 20$

- **A few examples**:

<table>
<thead>
<tr>
<th>decimal value</th>
<th>sign</th>
<th>truncated value</th>
<th>fractional portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>000000000010</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>-5</td>
<td>1</td>
<td>000000000101</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>000000000000</td>
<td>10000000000000000000</td>
</tr>
<tr>
<td>-9.125</td>
<td>1</td>
<td>000000001001</td>
<td>00100000000000000000</td>
</tr>
</tbody>
</table>

- **Range of values that can be represented**: 
  $$\pm 2^{12}(1 - 2^{-32})$$
Rounding errors in fixed precision computations

- Not all real numbers in the right range can be represented. Rounding is unavoidable.
  - This may happen both for basic operations and for program constants...

- **Example:** fraction $\frac{1}{10}$
  - $\frac{1}{10}$ cannot be represented exactly in fixed precision arithmetic.
  - Let us decompose $\frac{1}{10}$ as a sum of terms of the form $\frac{1}{2^i}$:
    $$\frac{1}{10} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \left( \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \cdot \frac{1}{5} \right) = \ldots$$
  - Infinite binary representation: $0.00011001100110011001100\ldots$
  - If $p = 24$:
    - Representation: “0.000110011001100110011001”
    - Rounding error is $9.5 \cdot 10^{-8}$

- **Floating precision numbers** (more commonly used today) have the same limitation

\[\text{Xavier Rival} \quad \text{Introduction} \quad 24 / 80\]
The root cause: a clock drift

Trajectory confirmation algorithm (summary):

- hardware clock $T_d$ ticks every tenth of a second
- time $T_c$ is computed in seconds: $T_c = \frac{1}{10} \times T_d$
- in binary: $T_c = 0.000110011001100110011001b \times b T_d$
- relative error is $10^{-6}$
- after the computer has been running for 100 h:
  - the absolute error is 0.34 s
  - as a “Scud” travels at 1700 m/s: the predicted window is about 580 m from where it should be
  - this explains the trajectory confirmation failure!

Remarks:

- the issue was discovered by israeli users, who noticed the clock drift
- their solution: frequently restart the control computer... (daily)
- this was not done in Dahran... the system had been running for 4 days
Precision issues in the fixed precision arithmetic:

- A scalar **constant** used in the code was **invalid**
  i.e., bound to be rounded to an approximate value, incurring a
  significant approximation the designers were unaware of

- There was **no adequate study of the precision** achieved by the
  system, although precision is clearly critical here!

**Current status:** such issues can be found by static analysis tools
Outline

1 Introduction

2 Case studies
   - Patriot missile (anti-missile system), Dahran (1991)
   - General remarks

3 Approaches to verification

4 Orderings, lattices, fixpoints
Common issues causing software problems

The examples given so far are not isolated cases
See for instance www.cs.tau.ac.il/~nachumd/horror.html (not up-to-date)

Typical reasons:

- **Improper specification** or understanding of the environment, conditions of execution...

- **Incorrect implementation of a specification**
  e.g., the code should be free of runtime errors
  e.g., the software should produce a result that meets some property

- **Incorrect understanding of the execution model**
  e.g., generation of too imprecise results
New challenges to ensure embedded systems do not fail

**Complex software architecture:** e.g. parallel softwares
- single processor multi-threaded, distributed (several computers)
- more and more common: multi-core architectures
- very hard to reason about
  - other kinds of issues: dead-locks, races...
  - very complex execution model: interleavings, memory models

**Complex properties to ensure:** e.g., security
- the system should resist even in the presence of an attacker (agent with malicious intentions)
- attackers may try to access sensitive data, to corrupt critical data...
- security properties are often even hard to express
Techniques to ensure software safety

Software development techniques:

- **software engineering**, with a focus on specification, and software quality (may be more or less formal...)
- **programming rules** for specific areas (e.g., DO 178 c in avionics)
- usually do not guarantee any strong property, but make softwares “cleaner”

Formal methods:

- should have **sound mathematical foundations**
- should allow to **guarantee** softwares meet some complex properties
- should be **trustable** (is a paper proof ok ???)
- **increasingly used in real life applications**, but still a lot of open problems
What is to be verified?

What do the programs below do?

\[ P_0 \]

```c
int x = 0;

int f_0(int y){
    return y * x;
}

int f_1(int y){
    x = y;
    return 0;
}

void main(){
    z = f_0(10) + f_1(100);
}
```

\[ P_1 \]

```c
void main(){
    int i;
    int t[100] = {0, 1, 2, ..., 99};
    while(i < 100){
        t[i] ++;
        i ++;
    }
}
```

\[ P_2 \]

```c
void main(){
    float f = 0.0;
    for(int i = 0; i < 1 000 000; i ++){
        f = f + 0.1;
    }
}
```
Semantic subtleties...

### $P_0$

```c
int x = 0;

int f_0(int y){
    return y * x;
}

int f_1(int y){
    x = y;
    return 0;
}

void main(){
    z = f_0(10) + f_1(100);
}
```

**Execution order:**
- **not specified in C**
- **specified in Java**
- if left to right, $z = 0$
- if right to left, $z = 1000$
Semantic subtleties...

**Initialization:**
- runtime error in Java
- read of a random value in C
  (the value that was stored before)

**Floating point semantics:**
- 0.1 is not representable exactly
  what is it rounded to by the compiler?
- rounding errors
  what is the rounding mode at runtime?
The two main parts of this course

1. **Semantics**
   - allow to *describe precisely the behavior of programs*
     should account for execution order, initialization, scope...
   - allow to *express the properties to verify*
     several important families of properties: safety, liveness, security...
   - also important to *transform and compile* programs

2. **Verification**
   - aim at *proving* semantic properties of programs
   - a very strong limitation: *indecidability*
   - *several approaches*, that make various compromises around indecidability
Outline

1. Introduction
2. Case studies
3. Approaches to verification
   - Indecidability and fundamental limitations
   - Approaches to verification
   - Summing up
4. Orderings, lattices, fixpoints
The termination problem

**Termination**

Program $P$ terminates on input $X$ if and only if any execution of $P$, with input $X$ eventually reaches a final state.

- **Final state:** final point in the program (i.e., not error)
- **We may want to ensure termination:**
  - processing of a task, such as, e.g., printing a document
  - computation of a mathematical function
- **We may want to ensure non-termination:**
  - operating system
  - device drivers

**The termination problem**

Can we find a program $P_t$ that takes as arguments a program $P$ and data $X$ and that returns “TRUE” if $P$ terminates on $X$ and “FALSE” otherwise?
The termination problem is not computable

- **Proof by reductio ad absurdum**, using a *diagonal argument*
  We assume there exists a program $P_a$ such that:
  - $P_a$ always terminates
  - $P_a(P, X) = 1$ if $P$ terminates on input $X$
  - $P_a(P, X) = 0$ if $P$ does not terminate on input $X$

- We consider the following program:

```c
void P0(P){
    if(Pa(P, P) == 1){
        while(TRUE){}    //loop forever
    }else{
        return;    //do nothing...
    }
}
```

- What is the return value of $P_a(P_0, P_0)$? i.e., $P_0$ does it terminate on input $P_0$?
The termination problem is not computable

- **What is the return value of** $P_a(P_0,P_0)$?  
  We know $P_a$ always terminates and returns either 0 or 1 (assumption). Therefore, we need to consider only two cases:
  - if $P_a(P_0,P_0)$ returns 1, then $P_0(P_0)$ loops forever, thus $P_a(P_0,P_0)$ should return 0, so we have reached a **contradiction**
  - if $P_a(P_0,P_0)$ returns 0, then $P_0(P_0)$ terminates, thus $P_a(P_0,P_0)$ should 1, so we have reached a **contradiction**

- In both cases, we **reach a contradiction**
- Therefore we conclude that **no such a** $P_a$ **exists**

The termination problem is not decidable

There exists no program $P_t$ that always terminates and always recognizes whether a program $P$ terminates on input $X$
Can we find a program $P_c$ that takes a program $P$ and input $X$ as arguments, always terminates and returns
- 1 if and only if $P$ runs safely on input $X$, i.e., without a runtime error
- 0 if $P$ crashes on input $X$

Answer: **No**, the same diagonal argument applies

**Non-computability result**
The absence of runtime errors is not computable
Rice theorem

- **Semantic specification**: set of *correct* program executions
- “Trivial” specifications:
  - empty set
  - set of all possible executions
  
  ⇒ intuitively, the non interesting verification problems...

Rice theorem (1953)

Considering a Turing complete language, any non trivial specification is not computable

- **Intuition**: there is no algorithm to decide non trivial specifications, starting with only the program code
- Therefore all interesting properties are **not computable**: termination, absence of runtime errors, absence of arithmetic errors, etc...
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Case studies</td>
</tr>
<tr>
<td>3</td>
<td>Approaches to verification</td>
</tr>
<tr>
<td></td>
<td>Indecidability and fundamental limitations</td>
</tr>
<tr>
<td></td>
<td>Approaches to verification</td>
</tr>
<tr>
<td></td>
<td>Summing up</td>
</tr>
<tr>
<td>4</td>
<td>Orderings, lattices, fixpoints</td>
</tr>
</tbody>
</table>
Towards partial solutions

- The initial verification problem is not computable

**Solution:** solve a weaker problem

**Several compromises can be made:**

- **simulation / testing:** observe only finitely many finite executions
  infinite system, but only finite exploration (no proof beyond that)
- **assisted theorem proving:** we give up on automation
  (no proof inference algorithm in general)
- **model checking:** we consider only finite systems
  (with finitely many states)
- **bug-finding:** search for “patterns” indicating “likely errors”
  (may miss real program errors, and report non existing issues)
- **static analysis based on abstraction:** attempt at automatic correctness proofs
  (yet, may fail to verify some correct programs)
Safety verification method characteristics

Safety verification problem

- **Semantics** $[P]$ of program $P$: set of behaviors of $P$ (e.g., states)
- **Property to verify** $S$: set of admissible behaviors (e.g., safe states)

Then, the verification problem boils down to showing:

$$[P] \subseteq S$$

- **Automation**: existence of an algorithm
- **Scalability**: should allow to handle large softwares
- **Soundness**: identify any wrong program
- **Completeness**: accept all correct programs
- **Apply to program source code**, i.e., not require a modeling phase
Approaches to verification

Testing by simulation

**Principle**

Run the program on *finitely many finite inputs*

- **Very widely used:**
  - **unit testing:** each function is tested separately
  - **integration testing:** with all surrounding systems, hardware (e.g., *iron bird* in avionics)

- **Automated**

- **Complete:** will never raise a false alarm

- **Unsound** unless exhaustive: *may miss program defects*

- **Costly:** needs to be re-done when software gets updated

Will not be studied in the course, classical development technique
Machine assisted proof

**Principle**

Have a **machine checked** proof, that is partly **human written**

- **tactics / solvers** may help in the inference
- the **hardest invariants** have to be user-supplied

**Applications**

- software industry (rare): Line 14 in Paris Subway
- hardware: ACL 2
- academia: CompCert compiler, SEL4 verified micro-kernel

**Not fully automated**

often turns out **costly** as complex proof arguments have to be found

**Sound** and **complete**
Model-Checking

Principle

Consider **finite systems** only, using algorithms for
- exhaustive exploration,
- symmetry reduction...

**Applications:**
- **hardware** verification
- **driver protocols** verification (Microsoft)

Applies on a **model**: a model extraction phase is needed
- for infinite systems, this is necessarily approximate
- not always automated

Automated, sound, complete with respect to the model
“Bug finding”

**Principle**

Identify “**likely**” issues, i.e., patterns known to often indicate an error

- **Example**: Coverity
- **Automated**
- **Not complete**: may report false alarms
- **Not sound**: may accept false programs
  thus **inadequate** for safety-critical systems
Static analysis with abstraction (1/4)

**Principle**

*Use some approximation, but always in a conservative manner*

- **Under-approximation** of the property to verify: $S_{\text{under}} \subseteq S$
- **Over-approximation** of the semantics: $[P] \subseteq [P]\text{upper}$
- We let an automatic static analyzer attempt to prove that:
  
  $$[P]\text{upper} \subseteq S_{\text{under}}$$

  If it succeeds, $[P] \subseteq S$

- In practice, the static analyzer **computes** $[P]\text{upper}, S_{\text{under}}$
**Static analysis with abstraction (2/4)**

**Soundness**

The abstraction will catch **any incorrect program**

- If $\llbracket P \rrbracket \not\subseteq S$, then $\llbracket P \rrbracket_{\text{upper}} \not\subseteq S_{\text{under}}$

  since  
  \[
  \begin{cases}
  S_{\text{under}} & \subseteq S \\
  \llbracket P \rrbracket & \subseteq \llbracket P \rrbracket_{\text{upper}}
  \end{cases}
  \]
Static analysis with abstraction (3/4)

**Incompleteness**

The abstraction may fail to certify some correct programs

**Case of a false alarm:**

- program $P$ is **correct**
- but the static analysis **fails**
Incompleteness

The abstraction may fail to certify some correct programs

- In the following case, the analysis cannot conclude anything

- One goal of the static analyzer designer is to avoid such cases

Static analysis using abstraction

- **Automatic**: $[P]_{upper}$, $S_{under}$ computed automatically
- **Sound**: reports any incorrect program
- **Incomplete**: may reject correct programs
Outline

1. Introduction

2. Case studies

3. Approaches to verification
   - Indecidability and fundamental limitations
   - Approaches to verification
   - Summing up

4. Orderings, lattices, fixpoints
## Approaches to verification

### Summing up

A summary of common verification techniques

<table>
<thead>
<tr>
<th></th>
<th>Automatic</th>
<th>Sound</th>
<th>Complete</th>
<th>Source level</th>
<th>Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>Yes</td>
<td>No ¹</td>
<td>Yes</td>
<td>Yes</td>
<td>sometimes ²</td>
</tr>
<tr>
<td>Assisted proving</td>
<td>No</td>
<td>Yes</td>
<td>Almost</td>
<td>No</td>
<td>sometimes ³</td>
</tr>
<tr>
<td>Model-checking</td>
<td>Yes</td>
<td>Yes</td>
<td>Partially ⁴</td>
<td>No</td>
<td>sometimes</td>
</tr>
<tr>
<td>Bug-finding</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>sometimes</td>
</tr>
<tr>
<td>Static analysis</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>sometimes</td>
</tr>
</tbody>
</table>

- Obviously, no approach checks all characteristics
- Scalability is a challenge for all

¹ unless full testing is doable
² full testing usually not possible except for small programs with finite state space
³ quickly requires huge manpower
⁴ only with respect to the finite models... but not with respect to infinite semantics
Outline

1 Introduction

2 Case studies

3 Approaches to verification

4 Orderings, lattices, fixpoints
   - Basic definitions on orderings
   - Operators over a poset and fixpoints
Order relations

Very useful in semantics and verification:

- **logical ordering**, expresses **implication** of logical facts
- **computational ordering**, useful to establish well-foundedness of fixpoint definitions, and for termination

**Definition: partially ordered set (poset)**

Let a set $S$ and a binary relation $\sqsubseteq \subseteq S \times S$ over $S$. Then, $\sqsubseteq$ is an **order relation** (and $(S, \sqsubseteq)$ is called a **poset**) if and only if it is

- **reflexive**: $\forall x \in S, \ x \sqsubseteq x$
- **transitive**: $\forall x, y, z \in S, \ x \sqsubseteq y \land y \sqsubseteq z \implies x \sqsubseteq z$
- **antisymmetric**: $\forall x, y \in S, \ x \sqsubseteq y \land y \sqsubseteq x \implies x = y$

- **notation**: $x \sqsubset y ::= (x \sqsubseteq y \land x \neq y)$
We often use **Hasse diagrams** to represent posets:

**Extensive definition:**
- $S = \{x_0, x_1, x_2, x_3, x_4\}$
- $\sqsubseteq$ defined by:

  \[
  \begin{align*}
  x_0 & \sqsubseteq x_1 \\
  x_1 & \sqsubseteq x_2 \\
  x_1 & \sqsubseteq x_3 \\
  x_2 & \sqsubseteq x_4 \\
  x_3 & \sqsubseteq x_4 \\
  \end{align*}
  \]
Example: semantics of automata

We consider the classical notion of **finite automata** and let

- $L$ be a finite set of **letters**
- $Q$ be a finite set of **states**
- $q_i, q_f \in Q$ denote the **initial** state and **final** state
- $\rightarrow \subseteq Q \times L \times Q$ be a **transition relation**

Then, the set of words recognized by $A = (Q, q_i, q_f, \rightarrow)$ is defined by:

$$L[A] = \{a_0a_1 \ldots a_n \mid \exists q_0 \ldots q_{n-1} \in Q, \; q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q_f\}$$
Example: automata and semantic properties

A simple automaton:

\[ L = \{ a, b \} \quad Q = \{ q_0, q_1, q_2 \} \]

\[ q_i = q_0 \quad q_f = q_2 \]

\[ q_0 \xrightarrow{a} q_1 \quad q_1 \xrightarrow{b} q_2 \quad q_2 \xrightarrow{a} q_1 \]

A few semantic properties:

- \( \mathcal{P}_1 \): no recognized word contains two consecutive \( b \)

\[ \mathcal{L}[A] \cap L^* bbL^* = \emptyset \]

- \( \mathcal{P}_0 \): all recognized words contain at least one occurrence of \( a \)

\[ \mathcal{L}[A] \subseteq L^* aL^* \]
Total ordering

Definition: total order relation

Order relation \( \sqsubseteq \) over \( S \) is a total order if and only if

\[
\forall x, y \in S, \ x \sqsubseteq y \lor y \sqsubseteq x
\]

- \((\mathbb{R}, \leq)\) is a total ordering
- if set \( S \) has at least two distinct elements \( x, y \) then its powerset \((\mathcal{P}(S), \subseteq)\) is not a total order
  - indeed \( \{x\}, \{y\} \) cannot be compared
- most of the order relations we will use are not be total
Minimum and maximum elements

Definition: extremal elements

Let \((S, \sqsubseteq)\) be a poset and \(S' \subseteq S\). Then \(x\) is

- **minimum element** of \(S'\) if and only if \(x \in S' \land \forall y \in S', \ x \sqsubseteq y\)
- **maximum element** of \(S'\) if and only if \(x \in S' \land \forall y \in S', \ y \sqsubseteq x\)

- maximum and minimum elements **may not exist**
  - example: \(\{\{x\}, \{y\}\}\) in the powerset, where \(x \neq y\)
- **infimum** \(\bot\) ("bottom"): minimum element of \(S\)
- **supremum** \(\top\) ("top"): maximum element of \(S\)
Upper bounds and least upper bound

**Definition: bounds**

Given poset \((S, \sqsubseteq)\) and \(S' \subseteq S\), then \(x \in S\) is

- **an upper bound** of \(S'\) if
  \[
  \forall y \in S', \ y \sqsubseteq x
  \]

- **the least upper bound** (lub) of \(S'\) (noted \(\sqcup S'\)) if
  \[
  \forall y \in S', \ y \sqsubseteq x \land \forall z \in S, (\forall y \in S', \ y \sqsubseteq z) \implies x \sqsubseteq z
  \]

- if it exists, the least upper bound is **unique**: if \(x, y\) are least upper bounds of \(S\), then \(x \sqsubseteq y\) and \(y \sqsubseteq x\), thus \(x = y\) by antisymmetry

- **notation:** \(x \sqcup y := \sqcup\{x, y\}\)

- *upper bounds and least upper bounds may not exist*

- **dual notions:** lower bound, greatest lower bound (glb, noted \(\sqcap S'\))
Duality principle

So far all definitions admit a symmetric counterpart

- given an order relation $\sqsubseteq$, $\mathcal{R}$ defined by $x \mathcal{R} y \iff y \sqsubseteq x$ is also an order relation
- thus all properties that can be proved about $\sqsubseteq$ also have a symmetric property that also holds

This is the **duality principle**:

| minimum element | maximum element |
| infimum         | supremum       |
| lower bound     | upper bound    |
| greatest lower bound | least upper bound |

... more to follow
Complete lattice

Definition: complete lattice

A **complete lattice** is a tuple \((S, \subseteq, \bot, \top, \sqcup, \sqcap)\) where:
- \((S, \subseteq)\) is a poset
- \(\bot\) is the infimum of \(S\)
- \(\top\) is the supremum of \(S\)
- any subset \(S'\) of \(S\) has a lub \(\sqcup S'\) and a glb \(\sqcap S'\)

**Properties:**
- \(\bot = \sqcup \emptyset = \sqcap S\)
- \(\top = \sqcap \emptyset = \sqcup S\)

**Example:** the powerset \((\mathcal{P}(S), \subseteq, \emptyset, S, \cup, \cap)\) of set \(S\) is a complete lattice
Lattice

The existence of lubs and glbs for all subsets is often a very strong property, that may not be met:

**Definition: lattice**

A **lattice** is a tuple \((S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) where:

- \((S, \sqsubseteq)\) is a poset
- \(\bot\) is the infimum of \(S\)
- \(\top\) is the supremum of \(S\)
- any pair \(\{x, y\}\) of \(S\) has a lub \(x \sqcup y\) and a glb \(x \sqcap y\)

- let \(Q = \{q \in \mathbb{Q} \mid 0 \leq q \leq 1\}\); then \((Q, \leq)\) is a **lattice** but **not a complete lattice**
- indeed, \(\{q \in Q \mid q \leq \frac{\sqrt{2}}{2}\}\) has no lub in \(Q\)
- property: a **finite** lattice is also a complete lattice
Chains

**Definition: increasing chain**

Let \((S, \sqsubseteq)\) be a poset and \(C \subseteq S\).

It is an **increasing chain** if and only if

- it has an infimum
- poset \((C, \sqsubseteq)\) is total (i.e., any two elements can be compared)

**Example**, in the powerset \((\mathcal{P}(\mathbb{N}), \subseteq)\):

\[
C = \{c_i \mid i \in \mathbb{N}\} \quad \text{where} \quad c_i = \{2^0, 2^2, \ldots, 2^i\}
\]

**Definition: increasing chain condition**

Poset \((S, \sqsubseteq)\) **satisfies the increasing chain condition** if and only if any increasing chain \(C \subseteq S\) is finite.
Complete partial orders

Definition: complete partial order

A **complete partial order** (cpo) is a poset \((S, \sqsubseteq)\) such that any increasing chain \(C\) of \(S\) has a least upper bound. A **pointed cpo** is a cpo with an infimum \(\bot\).

- clearly, any complete lattice is a cpo
- the opposite is not true:
Outline

1. Introduction
2. Case studies
3. Approaches to verification
4. Orderings, lattices, fixpoints
   - Basic definitions on orderings
   - Operators over a poset and fixpoints
Towards a constructive definition of the automata semantics

We now look for a constructive version of the automaton semantics as hinted by the following observations

**Observation 1:** \( \mathcal{L}[\mathcal{A}] = [\mathcal{A}](q_f) \) where

\[
[\mathcal{A}] : \quad Q \longrightarrow \mathcal{P}(L^*) \\
q \longmapsto \quad \{ w \in L^* \mid \exists n, \ w = a_0a_1 \ldots a_n \\
\exists q_0 \ldots q_{n-1} \in Q, \ q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q \}
\]

**Observation 2:** \( [\mathcal{A}] = \bigcup [\mathcal{A}]_n \) where

\[
[\mathcal{A}] : \quad Q \longrightarrow \mathcal{P}(L^*) \\
q \longmapsto \quad \{ a_0a_1 \ldots a_n \mid \exists q_0 \ldots q_{n-1} \in Q, \ q_i \xrightarrow{a_0} q_0 \xrightarrow{a_1} q_1 \ldots q_{n-1} \xrightarrow{a_n} q \}
\]

**Observation 3:** \( [\mathcal{A}]_{n+1} \) can be computed directly from \( [\mathcal{A}]_n \)

\[
[\mathcal{A}]_{n+1}(q) = \bigcup_{q' \in Q} \{ wa \mid w \in [\mathcal{A}]_n(q') \land q' \xrightarrow{a} q \}
\]
Operators over a poset

Definition: operators and orderings

Let \((S, \sqsubseteq)\) be a poset and \(\phi : S \rightarrow S\) be an operator over \(S\). Then, \(\phi\) is:

- **monotone** if and only if \(\forall x, y \in S, \ x \sqsubseteq y \implies \phi(x) \sqsubseteq \phi(y)\)
- **continuous** if and only if, for any chain \(S' \subseteq S\) then:
  \[
  \begin{cases}
  \text{if } \bigsqcup S' \text{ exists, so does } \bigsqcup \{\phi(x) \mid x \in S'\} \\
  \text{and } \phi(\bigsqcup S') = \bigsqcup \{\phi(x) \mid x \in S'\}
  \end{cases}
  \]
- **\(\sqsupseteq\)-preserving** if and only if:
  \[
  \forall S' \subseteq S, \ \begin{cases}
  \text{if } \bigsqcup S' \text{ exists, then } \bigsqcup \{\phi(x) \mid x \in S'\} \text{ exists} \\
  \text{and } \phi(\bigsqcup S') = \bigsqcup \{\phi(x) \mid x \in S'\}
  \end{cases}
  \]

Notes:

- “monotone” in English means “croissante” in French; “décroissante” translates into “anti-monotone” and “monotone” into “isotone”
- the dual of “monotone” is “monotone”
Operators over a poset

A few interesting properties:

- **continuous ⇒ monotone:**
  If \( \phi \) is monotone, and \( x, y \in S \) are such that \( x \sqsubseteq y \), then \( \{x, y\} \) is a chain with lub \( y \), thus \( \phi(x) \sqcup \phi(y) \) exists and is equal to 
  \( \phi(\sqcup\{x, y\}) = \phi(y) \); therefore \( \phi(x) \sqsubseteq \phi(y) \).

- **⊔-preserving ⇒ monotone:**
  same argument.
Fixpoints

Definition: fixpoints

Let \((S, \sqsubseteq)\) be a poset and \(f : S \rightarrow S\) be an operator over \(S\).

- a **fixpoint** of \(\phi\) is an element \(x\) such that \(\phi(x) = x\)
- a **pre-fixpoint** of \(\phi\) is an element \(x\) such that \(x \sqsubseteq \phi(x)\)
- a **post-fixpoint** of \(\phi\) is an element \(x\) such that \(\phi(x) \sqsubseteq x\)
- the **least fixpoint** \(\text{lfp}\phi\) of \(\phi\) (if it exists, it is unique) is the smallest fixpoint of \(\phi\)
- the **greatest fixpoint** \(\text{gfp}\phi\) of \(\phi\) (if it exists, it is unique) is the greatest fixpoint of \(\phi\)

**Note:** the existence of a least fixpoint, a greatest fixpoint or even a fixpoint is *not guaranteed*; we will see several theorems that establish their existence under specific assumptions...
Tarski’s Theorem

Theorem

Let \((S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) be a complete lattice and \(\phi : S \to S\) be a monotone operator over \(S\). Then:

1. \(\phi\) has a least fixpoint \(\text{lfp}\phi\) and \(\text{lfp}\phi = \sqcap\{x \in S \mid \phi(x) \sqsubseteq x\}\).
2. \(\phi\) has a greatest fixpoint \(\text{gfp}\phi\) and \(\text{gfp}\phi = \sqcup\{x \in S \mid x \sqsubseteq \phi(x)\}\).
3. the set of fixpoints of \(\phi\) is a complete lattice.

Proof of point 1:

We let \(X = \{x \in S \mid \phi(x) \sqsubseteq x\}\) and \(x_0 = \sqcap X\).

Let \(y \in X\):

- \(x_0 \sqsubseteq y\) by definition of the glb;
- thus, since \(\phi\) is monotone, \(\phi(x_0) \sqsubseteq \phi(y)\);
- thus, \(\phi(x_0) \sqsubseteq y\) since \(\phi(y) \sqsubseteq y\), by definition of \(X\).

Therefore \(\phi(x_0) \sqsubseteq x_0\), since \(x_0 = \sqcap X\).
Tarski’s Theorem

We proved that $\phi(x_0) \sqsubseteq x_0$. We derive from this that:

- $\phi(\phi(x_0)) \sqsubseteq \phi(x_0)$ since $\phi$ is monotone;
- $\phi(x_0)$ is a post-fixpoint of $\phi$, thus $\phi(x_0) \in X$;
- $x_0 \sqsubseteq \phi(x_0)$ by definition of the greatest lower bound.

We have established both inclusions so $\phi(x_0) = x_0$.

Proof of point 2: similar, by duality.

Proof of point 3:

- if $X$ is a set of fixpoints of $\phi$, we need to consider $\phi$ over $
\{y \in S \mid y \sqsubseteq S \cap X\}$ to establish the existence of a glb of $X$ in the poset of fixpoints
- the existence of least upper bounds in the poset of fixpoints follows by duality
Automata example, fixpoint definition

Lattice:

- \( S = Q \rightarrow \mathcal{P}(L^*) \)

- the ordering is the pointwise extension \( \sqsubseteq \) of \( \sqsubseteq \)

Operator:

- we let \( \phi_0 : S \rightarrow S \) be defined by
  \[
  \phi_0(f) = \lambda(q \in Q) \cdot \bigcup_{q' \in Q} \{ wa \mid w \in \llbracket A \rrbracket_n(q') \land q' \xrightarrow{a} q \}
  \]

- we let \( \phi : S \rightarrow S \) by defined by
  \[
  \phi(f) = \lambda(q \in Q) \cdot \left\{ \begin{array}{ll}
  \{ \epsilon \} \cup \phi_0(f)(q_i) & \text{if } q = q_i \\
  \phi_0(f)(q) & \text{otherwise}
  \end{array} \right.
  \]

Proof steps to complete:

- the existence of \( \text{lfp}\phi \) follows from Tarski’s theorem

- the equality \( \text{lfp}\phi = \llbracket A \rrbracket \) can be established by induction and double inclusion... but there is a simpler way
Kleene’s Theorem

Tarski’s theorem guarantees existence of an lfp, but is not constructive.

**Theorem**

Let \((S, \sqsubseteq, \bot)\) be a pointed cpo and \(\phi : S \to S\) be a continuous operator over \(S\). Then \(\phi\) has a least fixpoint, and

\[
\text{lfp}\phi = \bigsqcup_{n \in \mathbb{N}} \phi^n(\bot)
\]

First, we prove the existence of the lub:

Since \(\phi\) is continuous, it is also monotone. We can prove by induction over \(n\) that \(\{\phi^n(\bot) \mid n \in \mathbb{N}\}\) is a chain:

- \(\phi^0(\bot) = \bot \sqsubseteq \phi(\bot)\) by definition of the infimum;
- if \(\phi^n(\bot) \sqsubseteq \phi^{n+1}(\bot)\), then
  \[\phi^{n+1}(\bot) = \phi(\phi^n(\bot)) \sqsubseteq \phi(\phi^{n+1}(\bot)) = \phi^{n+2}(\bot)\]

By definition of the cpo structure, the lub exists. We let \(x_0\) denote it.
Kleene’s Theorem

Secondly, we prove that it is a fixpoint of $\phi$:

Since $\phi$ is continuous, $\{\phi^{n+1}(\bot) \mid n \in \mathbb{N}\}$ has a lub, and

$$
\phi(x_0) = \phi(\sqcup \{\phi^n(\bot) \mid n \in \mathbb{N}\}) \\
= \{\sqcup \phi^{n+1}(\bot) \mid n \in \mathbb{N}\} \quad \text{by continuity of } \phi \\
= \bot \sqcup \{\sqcup \phi^{n+1}(\bot) \mid n \in \mathbb{N}\} \quad \text{by definition of } \bot \\
= x_0 \quad \text{by simple rewrite}
$$

Last, we show that it is the least fixpoint:

Let $x_1$ denote another fixpoint of $\phi$. We show by induction over $n$ that $\phi^n(\bot) \sqsubseteq x_1$:

- $\phi^0(\bot) = \bot \sqsubseteq x_1$ by definition of $\bot$;
- if $\phi^n(\bot) \sqsubseteq x_1$, then $\phi^{n+1}(\bot) \sqsubseteq \phi(x_1) = x_1$ by monotony, and since $x_1$ is a fixpoint.

By definition of the lub, $x_0 \sqsubseteq x_1$
Automata example, constructive

We can now state a **constructive definition** of the automaton semantics. Operator $\phi$ is defined by

$$\phi(f) = \lambda(q \in Q) \cdot \begin{cases} \{\varepsilon\} \cup \phi_0(f)(q_i) & \text{if } q = q_i \\ \phi_0(f)(q) & \text{otherwise} \end{cases}$$

**Proof steps:**

- $\phi$ is continuous
- thus, Kleene’s theorem applies so $\text{lfp}\phi$ exists and $\text{lfp}\phi = \bigcup_{n \in \mathbb{N}} \phi^n(\bot)$...
  ... this actually saves the double inclusion proof to establish that \(\llbracket A \rrbracket = \text{lfp}\phi\)

Furthermore, $\llbracket A \rrbracket = \bigcup_{n \in \mathbb{N}} \phi^n(\bot)$.

This fixpoint definition will be very useful to infer or verify semantic properties.
Duality principle

We can extend the duality notion:

<table>
<thead>
<tr>
<th>monotone</th>
<th>monotone</th>
</tr>
</thead>
<tbody>
<tr>
<td>anti-monotone</td>
<td>anti-monotone</td>
</tr>
<tr>
<td>post-fixpoint</td>
<td>pre-fixpoint</td>
</tr>
<tr>
<td>least fixpoint</td>
<td>greatest fixpoint</td>
</tr>
<tr>
<td>increasing chain</td>
<td>decreasing chain</td>
</tr>
</tbody>
</table>

Furthermore both Tarski’s theorem and Kleene’s theorem have a dual version (Tarski’s theorem mostly encloses its own dual, except for the definition of the gfp).
In the next lectures...

- Families of **semantics**, for a general model of programs
- Families of **semantic properties of programs**
- **Verification techniques:**
  - abstract interpretation based static analysis
  - machine assisted theorem proving
  - model checking

**Next week:** transition systems and operational semantics
Practical information about the course

The course will be taught by:

- **Sylvain Conchon** (LRI, Paris-Orsay)
- **Antoine Miné** (DIENS)
- **Xavier Rival** (DIENS)

**Practical organization:**

- 1h30 Cours + 1h30 TD or TP depending on week
- a webpage will be available soon

**Evaluation:** \[ n = \frac{p + e}{2} \]

- **project** \( p \): implementation of a static analyzer
- **exam** \( e \): 28th of May, 2014