

# Merciless bug hunting

## why and how?

Simão Melo de Sousa

- the **need** for formal verification
- a study of some **examples** of **software errors**  
what are the causes ? what kind of properties do we want to verify ?
- a panel of the main **verification methods**  
with a fundamental limitation: **undecidability**
  - many techniques allow to **compute semantic properties**
  - each comes with **advantages** and **drawbacks**

---

## The need for Formal Verification

# What are *safety-critical* systems?

Systems that **cannot fail**

because of resulting

- loss of life
- threat on our well being
- loss of its purpose (mission critical)
- financial loss. etc.

example: medical devices, aircraft flight control, weapons, nuclear systems etc...

but also: digital money, digital transactions, online computer systems with sensible data, IoT, etc.

Our focus: critical systems that involve software

This is a very nice playground for the *Formal Verification* community

Software and cathedrals are much the same.  
First we build them, then we pray.

Sam Redwine, at the 4th International Software Process Workshop - 1988

The Formal Methods Approach:

**Aide toi, et le ciel t'aidera!**

Jean de La Fontaine - Le chartier embourbé. Livre VI - Fable 18.

... That is what this short introduction is about.


... That is what this short introduction is about.

Actually, a rather practical approach for this audience, but in contrast, keep in mind that software engineering is almost **pure handicraft**.

Any formal intake is indeed welcome.



(As a computer scientist, I'm allowed to imprecate upon Computer Science.)

 reliability  
correctness  
security  
safety  
robustness  
etc. . .

as a central challenge to the design of modern ICS.

- was and still is a central issue in the design of **safety critical systems**.
- The **reliability** quest lies at the foundations of Computer Science.



## Dysfunctions are expensive

- ICS maintenance =  $\frac{2}{3}$  of total costs;
- Dealing with dysfunctions or bugs = 20 times more expensive after than before production.
- Apart from these software life cycle considerations, it is well known that bugs can have deep impact in the customers' trust (e.g. Pentium bug) or financial losses (IBM vs FAA), letting aside human life losses (e.g. Therac 25) or ICS where bugs are simply unacceptable.

## About 16 years ago, W. Gibbs said:

Despite 50 years of progress, the software industry remains years – perhaps decades – short of the mature engineering discipline required to meet the needs of an information-age society.

*in: Trends in Computing: Software's Chronic Crisis - Scientific American - 1994.*

## Past? Provocation?

Unfortunately no. See for instance the usual **software equation**:

Software in the market = bug report channel

Interlude: Software Horror Stories([link](#))

Unfortunately,

**There is no definite solution!**

Nevertheless, there are satisfactory solutions:

**Reshape and Adapt** the software life cycle in order to include **reliability** as a central requirement.



**Common Criteria (EAL 5-7), Cenelec / IEC 61508 (Safety Integrity Level 3 and 4), DO-178B (Design Assurance Level A and B).**

Unfortunately,

**There is no definite solution!**

Nevertheless, there are satisfactory solutions:

integration and use of:

- Tests and Simulation
- **Formal Verification**

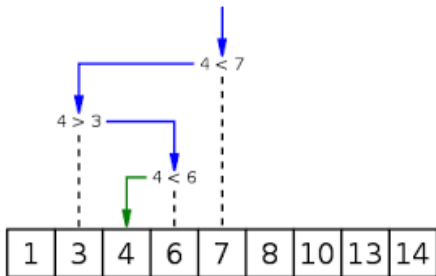
---

## Software Errors, and their consequences a few examples

## starting with a famous example

binary search: first publication in 1946

first publication **without bug** in 1962



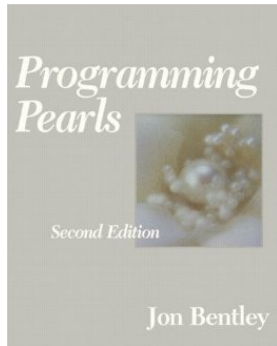
## starting with a famous example

Jon Bentley - Programming Pearls. 1986 (2<sup>nd</sup> ed. 2000)

(column 4) - writing correct programs  
The challenge of binary search

(as seen in p.37)

Warning  
Boring material ahead  
skip to section 4,4  
when drowsiness strikes



concise and clear explanation of the issue...

and yet...

in 2006, a bug was found in the Java standard library's binary search

Joshua Bloch, Google Research Blog

“Nearly All Binary Searches and Mergesorts are Broken”

it had been there for 9 years



the bug:

```
...  
    int mid = (low + high) / 2;  
    int midVal = a[mid];  
...
```

may exceed the capacity of type int

then provokes an access out of array bounds

```
int mid = low + (high - low) / 2
```

- **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...
- 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>

by 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 **regular flight data**
6. this causes the computation of an **absurd trajectory**
7. hence the **loss of control** of the launcher

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 i = ...;
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

## 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**

## 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

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

## 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>

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

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

Time	Distance from runway threshold	events
$T_0$	770 m	main landing gear landed
$T_0 + 3s$	1030 m	nose landing gear landed <b>brake command activated</b>
$T_0 + 12s$	1680 m	spoilers activated
$T_0 + 14s$	1800 m	thrust reversers activated
$T_0 + 31s$	2700 m	end of runway

# Protection of aircraft brake systems

Braking systems inhibition: **Prevent in-flight activation !**

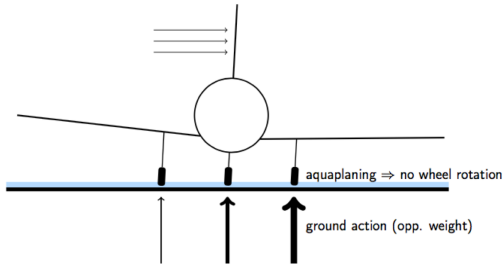
- spoilers: increase in aerodynamic load (drag)
- thrust reversers: could destroy the plane if activated in-flight ! (ex : crash of a B 767-300 ER Lauda Air, 1991, 223 fatalities; thrust reversers in-flight activation, electronic circuit issue)

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  
[Minimum Thrust] **AND** ([Weight] **OR** [Wheels spinning])

## Landing configuration:



## 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

[Minimum Thrust] AND ([Weight] OR [Wheels spinning])

# Flight 2904, a summary of the issues

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

- Old condition that failed to be satisfied:

$$(P_{left} > 6T)AND(P_{right} > 6T)$$

- Fixed condition (used in the new version of the software):

$$(P_{left} + P_{right}) > 12T$$

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

# The anti-missile “Patriot” system - Dahran (1991)

**Purpose:** destroy foe missiles before they reach their target

**Use in wars:**

- first Gulf war (1991)  
protection of towns and military facilities in Israel 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  $1700m/s$  ; travel about  $1000kms$  in 10 minutes
- not to destroy a friendly target (it happened at least twice!)
- very high cost: about \$1000000 per launch



## 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  $1500m/s$ )

**Very short process:** about ten minutes

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 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:

decimal value	sign	truncated value	fractional portion
2	0	00000000010	00000000000000000000
-5	1	00000000101	00000000000000000000
0.5	0	00000000000	10000000000000000000
-9.125	1	00000001001	00100000000000000000

- 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**

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  $\frac{1}{2^i}$

$$\begin{aligned}\frac{1}{10} &= \frac{1}{2} \times \frac{1}{5} \\ \frac{1}{5} &= \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \times \frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \times \left( \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \times \frac{1}{5} \right) = \dots\end{aligned}$$

- infinite binary representation: 0.00011001100110011001100...
- if  $p = 24$ : representation: "0.000110011001100110011001"
- rounding error is  $9.5 \times 10^{-8}$
- **Floating precision numbers** (more commonly used today) **have the same limitation**

## 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 as 1700m/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

# Patriot missile failure, a summary of the issues

**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

# 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](http://www.cs.tau.ac.il/~nachumd/horror.html) - Software Horror Stories

## 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/critical 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



## 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 B/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 ?

```
/*P0*/
```

```
int x = 0;
```

```
int f0(int y){  
    return y * x ;  
}
```

```
int f1 (int y){  
    x = y;  
    return 0;  
}
```

```
void main (){  
    z = f0(10) + f1(100);  
}
```

```
/*P1*/
```

```
void main (){  
    int i;  
    int t[100] = {0,1,2,3, ... ,99};  
    while(i<100){  
        t[i]++;  
        i++;  
    }  
}
```

```
/*P2*/
```

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

```
/*P0*/  
  
int x = 0;  
  
int f0(int y){  
    return y * x ;  
}  
  
int f1 (int y){  
x = y;  
return 0;  
}  
  
void main (){  
z = f0(10) + f1(100);  
}
```

Execution order:

**not specified in C**

**specified in Java**

if left to right,  $z = 0$

if right to left,  $z = 1000$

```
/*P1*/  
  
void main (){  
    int i;  
    int t[100] = {0,1,2,3, ... ,99};  
    while(i<100){  
        t[i]++;  
        i++;  
    }  
}
```

Initialization:

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

```
/*P2*/  
  
void main (){  
    float f = 0.;  
    for (int i = 0; i < 1 000 000; i++){  
        f = f + 0.1;  
    }  
}
```

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 ?

## 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

## Verification

- aim at **proving** semantic properties of programs
- a very strong limitation: **undecidability**
- **several approaches**, that make various compromises around undecidability

---

## a snapshot on Verification Methods

## 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 argument 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:

```
void  $P_0(P)$ {  
  if ( $P_a(P, P) = 1$ ) {  
    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 be 1, so we have, again, reached a **contradiction!**

In both cases, we **reach a contradiction** Therefore **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. if  $P_c(P, X)$  decides whether  $P$  will run safely on  $X$ , consider

```
void  $P_1(P)$ {  
  if ( $P_c(P, P) = 1$ ){  
    crash(); // fail (unsafe)  
  } else {  
    return; //do nothing (safe)  
  }  
}
```

**Non-computability result:**

**The absence of runtime errors is not computable**

- **Semantic specification:** set of *correct* program executions
  - “Trivial” semantic 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 semantic 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...

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 with abstraction:** attempt at automatic correctness proofs (yet, may fail to verify some correct programs)

## Safety verification problem

- **Semantics**  $\llbracket P \rrbracket$  of program  $P$ : set of behaviors of  $P$  (e.g., states)
- **Property to verify**  $\mathcal{S}$ : set of admissible behaviors (e.g., safe states)

**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 modelling phase

## Principle

Run the program on **finitely many finite inputs**

- maximize coverage
- inspect erroneous traces to fix bugs

## Very widely used:

- unit testing: each function is tested separately
- integration testing: with all surrounding systems, hardware

**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

## 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**

## 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**



## Principle

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

- use bounded symbolic execution, model exploration...
- rank "defect" reports using heuristics

Example: Coverity

## Automated

**Not complete:** may report false alarms

**Not sound:** may accept false programs thus inadequate for safety-critical systems

## Principle

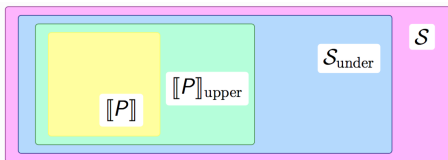
Use some approximation, **but always in a conservative manner**

**Under-approximation** of the property to verify:  $\mathcal{S}_{under} \subseteq \mathcal{S}$

**Over-approximation** of the semantics:  $\llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{upper}$

We let an automatic static analyzer attempt to prove that:  $\llbracket P \rrbracket_{upper} \subseteq \mathcal{S}_{under}$   
If it succeeds,  $\llbracket P \rrbracket \subseteq \mathcal{S}$

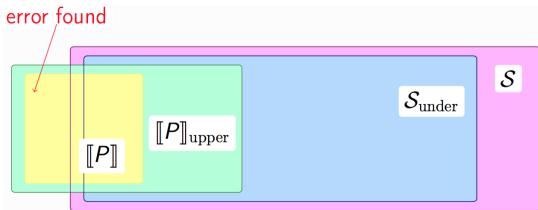
In practice, the static analyzer **computes**  $\llbracket P \rrbracket_{upper}, \mathcal{S}_{under}$



## Soundness

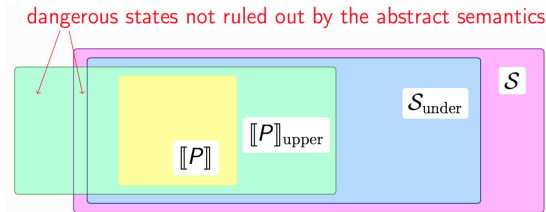
The abstraction will catch **any incorrect program**

if  $\llbracket P \rrbracket \not\subseteq \mathcal{S}$ , then  $\llbracket P \rrbracket_{upper} \not\subseteq \mathcal{S}_{under}$ , since  $\begin{cases} \mathcal{S}_{under} \subseteq \mathcal{S} \\ \llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{upper} \end{cases}$



## Incompleteness

The abstraction may fail to **certify some correct programs**



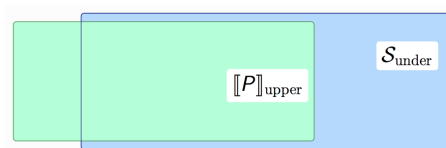
Case of a false alarm:

- 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:**  $\llbracket P \rrbracket_{\text{upper}}$ ,  $S_{\text{under}}$  computed automatically
- **Sound:** reports any incorrect program
- **Incomplete:** may reject correct programs

# A summary of common verification techniques

	Automatic	Sound	Complete	Source Level	Scalable
Simulation	YES	NO	YES	YES	sometimes
Assisted Proving	NO	YES	Almost	Partially	sometimes
Model-Checking	YES	YES	Partially	NO	sometimes
Bug-Finding	YES	NO	NO	YES	sometimes
Static Analysis	YES	YES	NO	YES	sometimes

- Obviously, no approach checks all characteristics
- Scalability is a challenge for all
- Note that the expressivity of Assisted Proof Approach come with a price: the loss of automation

Classical trade-off  $\Rightarrow$  **Expressivity versus Automation**

---

# On the use of Formal Verification Techniques in Critical Software Development

## Numerous examples, for instance:

- Garbage Collection, Operating System Modules, Communication and Cryptographic Protocols, Circuits and Hardware, Execution Platforms, Programming Language (semantics and methodologies), Compilers, Information Systems, etc... (see famous FM surveys for details)
- Application area (mostly safety critical systems):
  - Avionic and Aerospace (NASA, FAA, ARIANE, etc...),
  - Railways (subway, etc...)
  - Nuclear Systems
  - Medical Systems
  - (Real-Time) Embedded Systems



A quite old: The formal verification of the CISC project from IBM (Huxley Park- UK & Oxford): Update of an Information System.

- 800 000 lines of code. 268 000 were rewritten, 37 00 via the Z Formal Method (only Formal Spec. without proofs)
- results:
  - Development costs:  $-9\%$
  - 2.5 times fewer bugs, and the detected bugs were qualified as minor. (imply lower maintenance cost)

*First real success was Meteor line 14 driverless metro in Paris: Over 110 000 lines of B models were written, generating 86 000 lines of Ada. No bugs were detected after the proofs, neither at the functional validation, at the integration validation, at on-site test, nor since the metro lines operate (October 1998). The safety-critical software is still in version 1.0 in year 2007, without any bug detected so far.*

**In Formal Methods in Safety-Critical Railway Systems, Thierry Lecomte, Thierry Servat, Guilhem Pouzancre.**

# ASTRÉE Success story (excerpt from its web-site)

The development of ASTRÉE started from scratch in Nov. 2001. Two years later, the main applications have been the static analysis of synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language. ASTRÉE has achieved the following unprecedented results:

- In Nov. 2003, ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1h20 on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon 64 using 580 Mb of memory).
- From Jan. 2004 on, ASTRÉE was extended to analyze the electric flight control codes then in development and test for the A380 series. The operational application by Airbus France at the end of 2004 was just in time before the A380 maiden flight on Wednesday, 27 April, 2005.
- In April 2008, ASTRÉE was able to prove completely automatically the absence of any RTE in a C version of the automatic docking software of the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station.

## COQ and Common Criteria (excerpt from the official announcements)

- September 2007: a big step in program certification in the real world: The Technology and Innovation group at Gemalto has successfully completed a Common Criteria (CC) evaluation on a Java Card based commercial product. This evaluation is the world's first CC certificate of a Java product involving EAL7 components
- Trusted Logic announces (press release of November 18th, 2003) that the DCSSI has successfully evaluated its security methodology applied to the Java Card System at the Common Criteria EAL7 level, in a report published earlier this year. Coq is the proof engine used by Trusted Logics, and was chosen for its expressiveness. As a part of the certification process, it is being acknowledged as trustworthy by the DCSSI.

# Certified Compilation (excerpt from the **compcert** website)

Compcert is a compiler that generates PowerPC assembly code from Clight, a large subset of the C programming language. The particularity of this compiler is that it is written mostly within the specification language of the Coq proof assistant, and its correctness — the fact that the generated assembly code is semantically equivalent to its source program — was entirely proved within the Coq proof assistant.

A high-level overview of the Compcert compiler and its proof of correctness can be found in the following papers:

- Xavier Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. Proceedings of the POPL 2006 symposium.
- Sandrine Blazy, Zaynah Dargaye and Xavier Leroy, Formal verification of a C compiler front-end. Proceedings of Formal Methods 2006, LNCS 4085.

- Correctness of RSA Algorithm, by José C. Almeida (DIUM), Laurent Théry
- Certifying Prime Number with the Coq prover. CoqPrime is a library built on top of the Coq proof system to certify primality using Pocklington certificate and Elliptic Curve Certificate.
- SCALP Project: Security of Cryptographic ALgorithms with Probabilities. Probabilistic language and semantics for cryptographic proofs, Formalization of random generators, Proof theory for: High-level reasoning about distributions defined by probabilistic programs, Semantic preserving program transformations, Cryptography-based program transformations, Asymptotic reasoning.

## Java Card in COQ

Formal verification of the Java-Card Platform in COQ (joint work with G. Barthe, G. Dufay)

1. a specification and prototype of JavaCard Execution Platform and
2. the proof that (a R. Milner quote)

**Well-typed (JavaCard) Programs cannot go wrong**

3. A provably correct implementation of the ByteCode Verifier (BCV), a crucial security module based on static program analysis.

## SEL4

The L4.verified project, A Formally Correct Operating System Kernel (Gerwin Klein et al.)

- Goal : CENELEC SIL4 Railway Signaling System for the **Metro do Porto** - linha Aeroporto Sá Carneiro.
- Challenge: Software layer design , validation and certification (SIL4 - the highest) using Formal Methods in a very restrictive normative context (CENELEC). The first of its nature, to the best of our knowledge.
- Other Mission: set-up and training of a (formal) Validation team in a industrial context.
- Extension of the Scade toolset to deal with Function Block based HW.
  - a (pencil and paper proved) translation methodology and
  - a (HW level) testing framework with tests generated from SCADE models – allow for a better confidence on the translation process

(Highlight: First Signaling System **in the world!** formally and completely proved from scratch that reach the new CENELEC SIL4 certification)



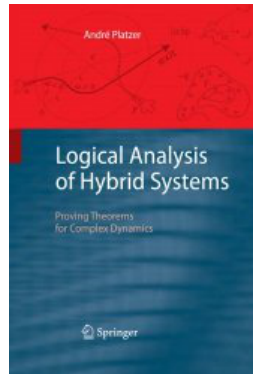
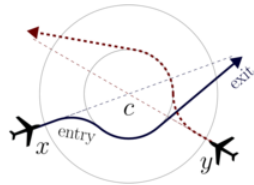
A very interesting case study for this audience:

the **safety** proof of the Airborne Collision Avoidance System ACAS X

using the proof calculus for differential algebraic dynamic logic and differential invariants

For a detailed account see

André Platzer. **Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics**. Springer, 2010. 426 p.  
and <http://symbolaris.com/info/RCAS.html>



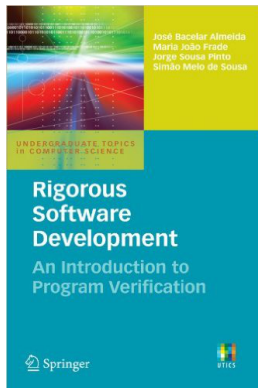
The book I use in my lectures on this subject (...why?)

## **Rigorous Software Development, A Practical Introduction to Program Verification**

Series: Undergraduate Topics in Computer Science.

Almeida, J.B., Frade, M.J., Pinto, J.S., Melo de Sousa, S.,  
Springer Verlag, 1st Edition., 2011, XIII, 307 p. 52 illus.  
Softcover, ISBN 978-0-85729-017-5

A more detailed account on the underlying concepts of  
Formal Methods and a comprehensive comparative survey  
can be found in the book.



a substencial part of the material exposed here was also taken from the Semantics and Applications to Verification course of the École Normale Superieure d'Ulm (X. Rival & A. Miné).

the binary search example was taken from Jean-Christophe Filliatre's slides on program verification

Un peu de programmation éloigne de la logique mathématique;  
beaucoup de programmation y ramène.

Xavier Leroy.

(a bit of programming moves the programmer away from logic, a lot of programming gets back to it)