Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Patrick Cousot and Radhia Cousot

École normale supérieure, Paris *École normale supérieure & CNRS, Paris*

joint work with:

Julien Bertrane
École normale supérieure, Paris

Jérôme Feret
École normale supérieure & INRIA, Paris

Laurent Mauborgne
École normale supérieure, Paris & IMDEA Software, Madrid

Antoine Miné
École normale supérieure & CNRS, Paris

Xavier Rival
École normale supérieure & INRIA, Paris

Workshop on formal verification of avionics software products

Airbus France, Toulouse, France June 24, 2010

Content

- Brief motivation
- An informal introduction to abstract interpretation
- A short overview of a few applications and on-going work at ENS on aerospace software
- A recent comprehensive overview paper (with all theoretical and practical details and references):

  J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné and X. Rival

  Static analysis and verification of aerospace software by abstract interpretation

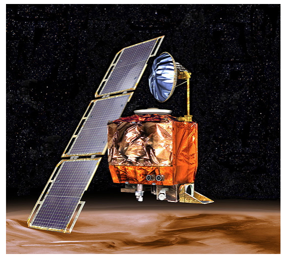
  AIAA Infotech@Aerospace 2010, Atlanta, Georgia, USA, April 20, 2010

Motivation

Computer scientists have made great contributions to the failure of complex systems

Ariane 5.01 failure (overflow)

Patriot failure (float rounding)



Mars orbiter loss (unit error)

- Checking the presence of bugs is great but never ends
- Proving their absence is even better!

Abstract interpretation

Abstract interpretation

- *Started in the 70's* and well-developped since then
- Originally for inferring program invariants (with first applications to compilation, optimization, program transformation, to help hand-made proofs, etc)
- Based on the idea that undecidability and complexity of automated program analysis can be fought by *approximation*
- Applications evolved from *static analysis* to *verification*
- ***Does scale up!***

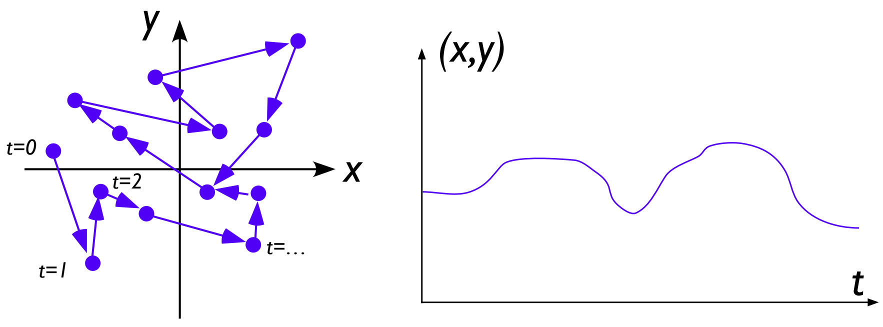
Fighting undecidability and complexity in program verification

- Any *automatic* program verification method will definitely fail on infinitely many programs (Gödel)
- Solutions:
  - Ask for human help (theorem-prover based *deductive methods*)
  - Consider (small enough) finite systems (*model-checking*)
  - Do sound approximations or complete abstractions (*abstract interpretation*)

An informal introduction to abstract interpretation

I) Define the programming language semantics

*Formalize the concrete **execution** of programs (e.g. transition system)*

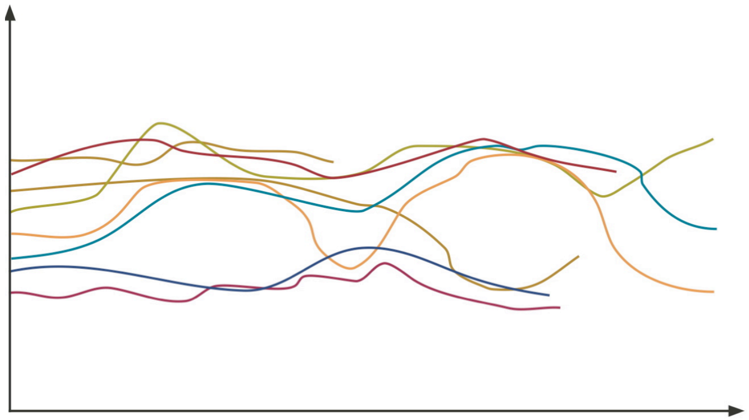


Trajectory
in state space

Space/time trajectory

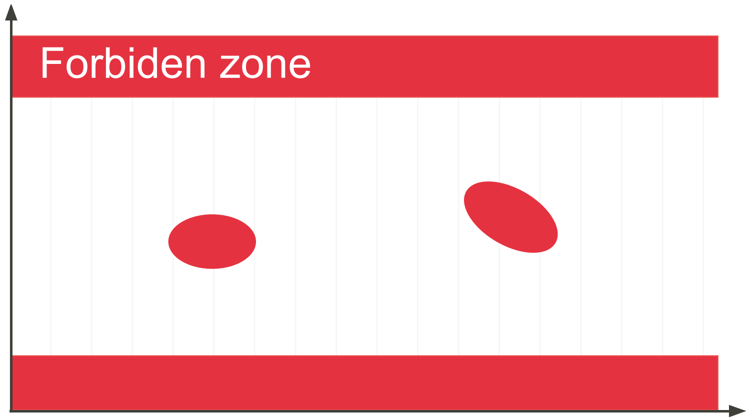
II) Define the program properties of interest

*Formalize what you are interested to **know** about program behaviors*



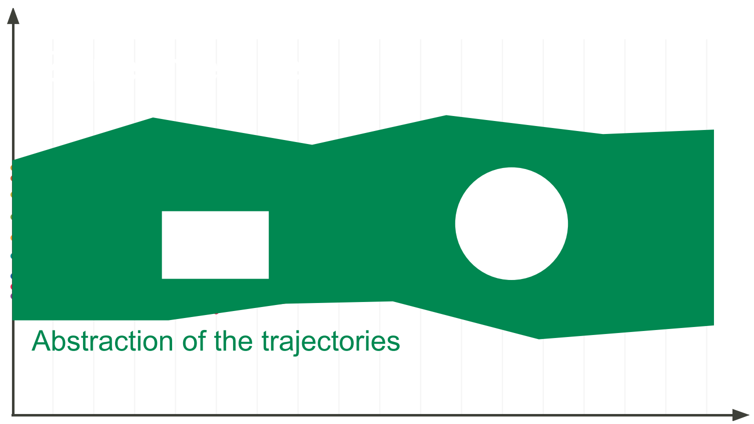
III) Define which specification must be checked

*Formalize what you are interested to **prove** about program behaviors*



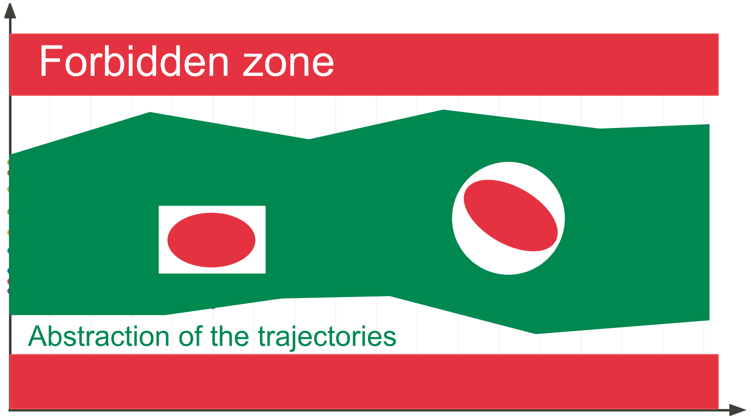
IV) Choose the appropriate abstraction

***Abstract** away all information on program behaviors irrelevant to the proof*



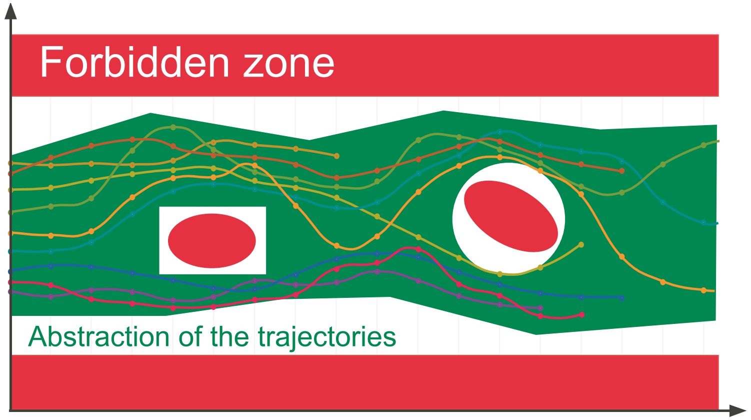
V) Mechanically verify in the abstract

*The proof is fully **automatic***



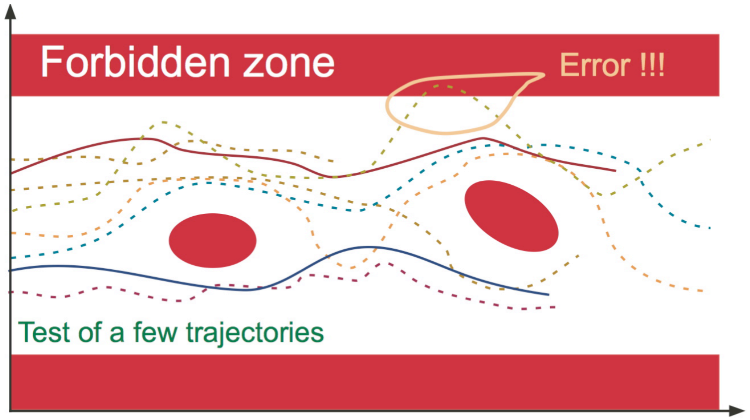
Soundness of the abstract verification

*Never forget any possible case so the **abstract proof is correct in the concrete***



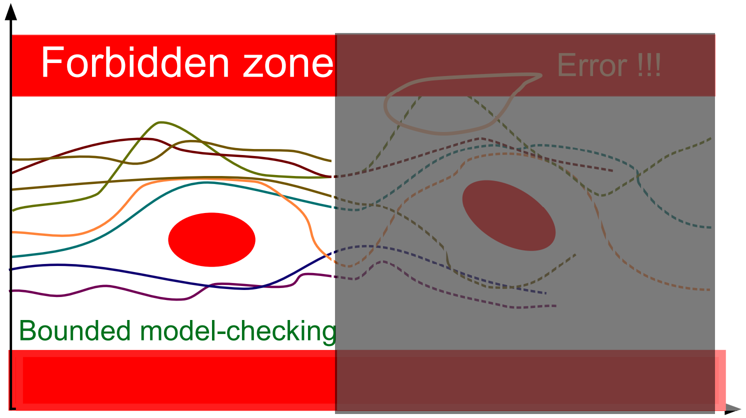
Unsound validation: testing

Try a few cases



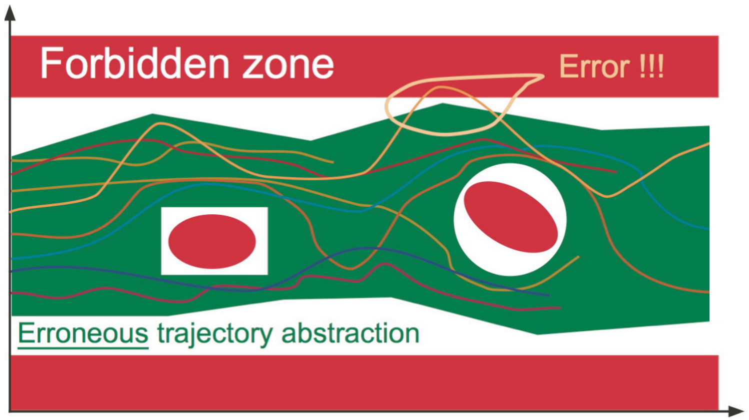
Unsound validation: bounded model-checking

Simulate the beginning of all executions



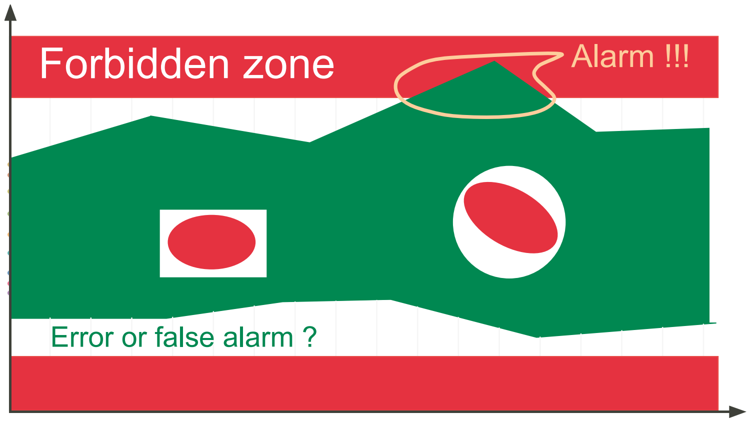
Unsound validation: static analysis

Many static analysis tools are ***unsound*** *(e.g. Coverity, etc.) so inconclusive*



Incompleteness

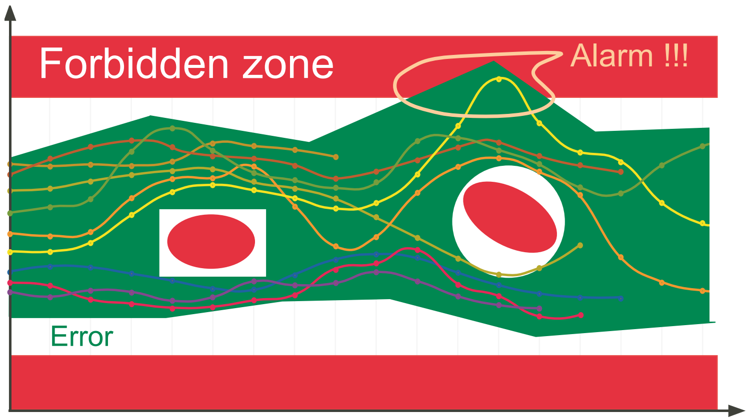
When abstract proofs may fail while concrete proofs would succeed



By soundness an alarm must be raised for this overapproximation!

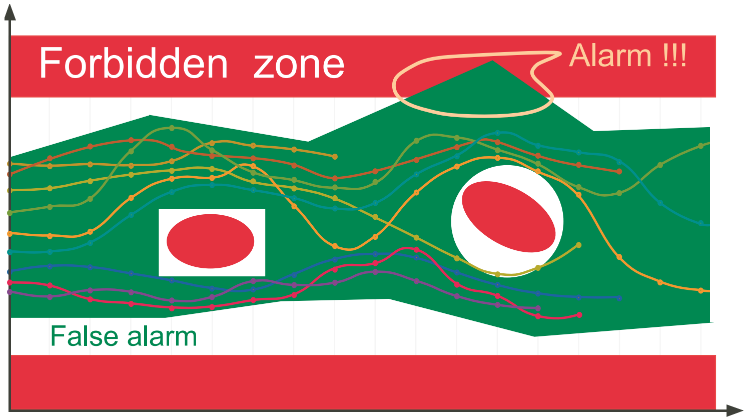
True error

The abstract alarm may correspond to a concrete error



False alarm

The abstract alarm may correspond to no concrete error (false negative)



What to do about false alarms?

- Automatic refinement: inefficient and may not terminate (Gödel)
- Domain-specific abstraction:
  - Adapt the abstraction to the *programming paradigms* typically used in given *domain-specific applications*
  - e.g. *synchronous control/command*: no recursion, no dynamic memory allocation, maximum execution time, etc.

ASTRÉE

Target language and applications

- C programming language
  - Without recursion, `longjump`, dynamic memory allocation, conflicting side effects, backward jumps, system calls (stubs)
  - With all its horrors (`union`, pointer arithmetics, etc)
  - Reasonably extending the standard (e.g. size & endianess of integers, IEEE 754-1985 floats, etc)
- Synchronous control/command
  - e.g. generated from Scade

The semantics of C implementations is very hard to define

What is the effect of out-of-bounds array indexing?

```
% cat unpredictable.c
#include <stdio.h>
int main () { int n, T[1];
 n = 2147483647;
 printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

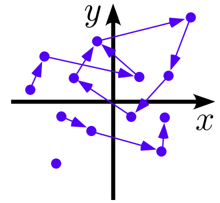
Yields different results on different machines:

`n = 2147483647, T[n] = 2147483647`	Macintosh PPC
`n = 2147483647, T[n] = -1208492044`	Macintosh Intel
`n = 2147483647, T[n] = -135294988`	PC Intel 32 bits
`Bus error`	PC Intel 64 bits

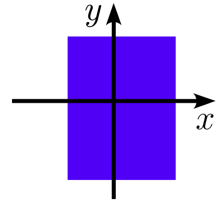
Implicit specification

- Absence of runtime errors: overflows, division by zero, buffer overflow, null & dangling pointers, alignment errors, …
- Semantics of runtime errors:
  - Terminating execution: stop (e.g. floating-point exceptions when traps are activated)
  - Predictable outcome: go on with worst case (e.g. signed integer overflows result in some integer, some options: e.g. modulo arithmetics)
  - Unpredictable outcome: stop (e.g. memory corruption)

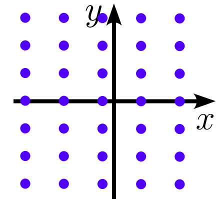
Abstractions



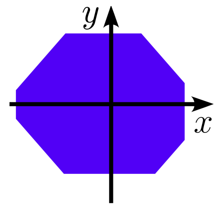
Collecting semantics:
partial traces



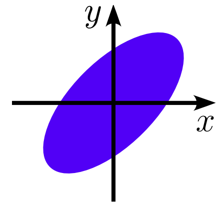
Intervals:
$\mathtt{x} \in [a, b]$



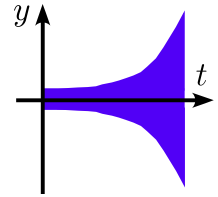
Simple congruences:
$\mathtt{x} \equiv a[b]$



Octagons:
$\pm\mathtt{x} \pm \mathtt{y} \leqslant a$



Ellipses:
$\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{x}\mathtt{y} \leqslant d$



Exponentials:
$-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

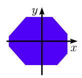
Example of general purpose abstraction: octagons

- Invariants of the form $\pm\,\mathtt{x} \pm \mathtt{y} \leq \mathtt{c}$, with $\mathcal{O}(\mathrm{N}^2)$ memory and $\mathcal{O}(\mathrm{N}^3)$ time cost.
- Example:

```
while (1) {
  R = A-Z;
  L = A;
  if (R>V)
    { ★ L = Z+V; }
  ★
}
```

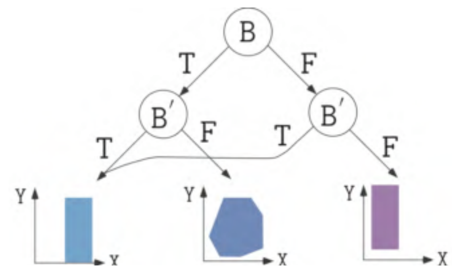
- At ★, the interval domain gives $\mathtt{L} \leq \max(\max \mathtt{A}, (\max \mathtt{Z}) + (\max \mathtt{V}))$.
- In fact, we have $\mathtt{L} \leq \mathtt{A}$.
- To discover this, we must know at ★ that $\mathtt{R} = \mathtt{A}\text{-}\mathtt{Z}$ and $\mathtt{R} > \mathtt{V}$.

- Here, $\mathtt{R} = \mathtt{A}\text{-}\mathtt{Z}$ cannot be discovered, but we get $\mathtt{L}\text{-}\mathtt{Z} \leq \max \mathtt{R}$ which is sufficient.
- We use many octagons on **small packs** of variables instead of a large one using all variables to cut costs.



Example of general purpose abstraction: decision trees

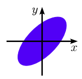
```
/* boolean.c */
typedef enum {F=0,T=1} BOOL;
BOOL B;
void main () {
  unsigned int X, Y;
  while (1) {
    ...
    B = (X == 0);
    ...
    if (!B) {
      Y = 1 / X;
    }
    ...
  }
}
```



The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leaves

Example of domain-specific abstraction: ellipses

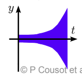
```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
             + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```



Example of domain-specific abstraction: exponentials

```
% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I; int R; BOOLEAN T;
void main() {
  R = 0;
  while (TRUE) {
    __ASTREE_log_vars((R));
    if (I) { R = R + 1; }
    else { R = 0; }
    T = (R >= 100);
    __ASTREE_wait_for_clock(());
  }}
% cat count.config
__ASTREE_volatile_input((I [0,1]));
__ASTREE_max_clock((3600000));
% astree -exec-fn main -config-sem count.config count.c|grep '|R|'
|R| <= 0. + clock *1. <= 3600001.
```

← potential overflow!

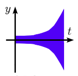


Example of domain-specific abstraction: exponentials

```
% cat retro.c
typedef enum {FALSE=0, TRUE=1} BOOL;
BOOL FIRST;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;

void dev( )
{ X=E;
  if (FIRST) { P = X; }
  else
    { P =  (P - ((((2.0 * P) - A) - B)
            * 4.491048e-03)); };
  B = A;
  if (SWITCH) {A = P;}
  else {A = X;}
}
```

```
void main()
{ FIRST = TRUE;
  while (TRUE) {
    dev( );
    FIRST = FALSE;
    __ASTREE_wait_for_clock(());
  }}
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));
|P| <= (15.  + 5.87747175411e-39
/ 1.19209290217e-07) * (1 +
1.19209290217e-07)^clock - 5.87747175411e-39
/ 1.19209290217e-07 <= 23.0393526881
```



An erroneous common belief on static analyzers

"The properties that can be proved by static analyzers are often simple" [2]

Like in mathematics:

- May be simple to state (no overflow)
- But harder to discover (`S[0], S[1] in [-1327.02698354, 1327.02698354]`)
- And difficult to prove (since it requires finding a non trivial non-linear invariant for second order filters with complex roots [Fer04], which can hardly be found by exhaustive enumeration)

Reference

[2] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 7, July 2008.

Industrial applications

Examples of applications

- Verification of the absence of runtime-errors in
  - Fly-by-wire flight control systems

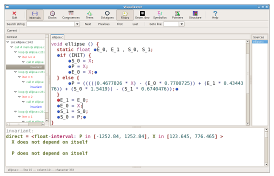
  - ATV docking system

  - Flight warning system (on-going work)

Industrialization

- 8 years of research (CNRS/ENS/INRIA):

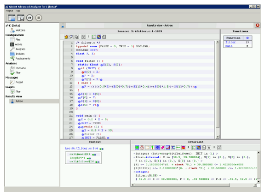
www.astree.ens.fr

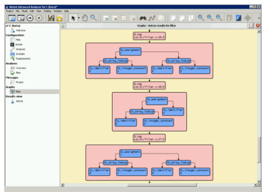


- Industrialization by AbsInt (since Jan. 2010):

www.absint.com/astree/

AbsInt Angewandte Informatik





On-going work

Verification of target programs

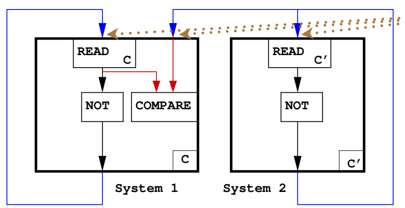
Verification of compiled programs

- The valid source may be proved correct while the certified compiler is incorrect so the target program may go wrong
- Possible approaches:
  - Verification at the target level
  - Source to target proof translation and proof check on the target
  - ✳ Translation validation (local verification of equivalence of run-time error free source and target)
  - Formally certified compilers

Verification of imperfectly clocked synchronous systems

Imperfect synchrony

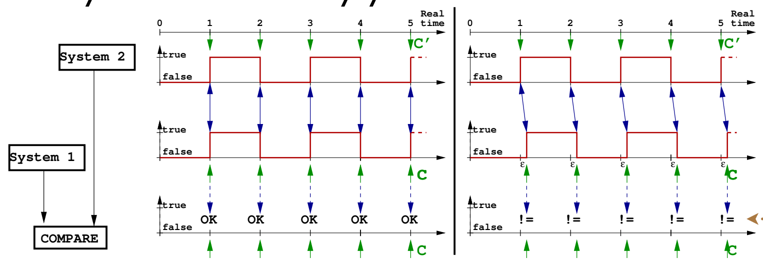
- Example of (buggy) communicating synchronous systems:



blackboard inputs

- negate previous input (on clocks C and C')
- compare inputs

- Synchronized and dysynchronized executions:



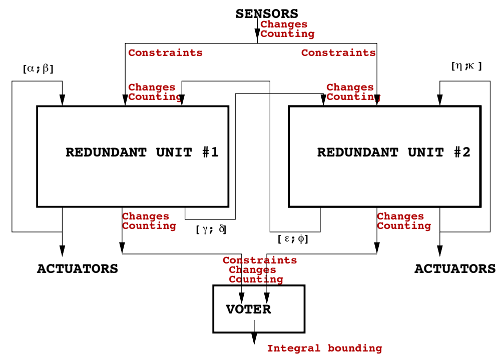
flawed alarms

Semantics and abstractions

- Continuous semantics (value *s*(*t*) of signals *s* at any time *t*)
- Clock ticks and serial communications do happen in known time intervals $[l, h], l \leq h$
- Examples of abstractions:
  - $\forall t \in [a;b] : s(t) = x.$
  - $\exists t \in [a;b] : s(t) = x.$
  - change counting $(\leqslant k, a \blacktriangleright \blacktriangleleft b)$ and $(\geqslant k, a \blacktriangleright \blacktriangleleft b)$

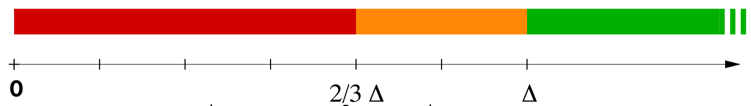
    (signal changes less (more) than *k* times in time interval $[a, b]$)

Example of static analysis



For how long should the input be stabilized before deciding on disagreement?

Specification : no alarm raised with a normal input



input stability $< \Delta$: counter-example	Between $\frac{2}{3} \times \Delta$ and Δ : ?	input stability $> \Delta$: the analyzer proves the specification

THÉSÉE: Verification of embedded real-time parallel C programs

Parallel programs

- Bounded number of processes with shared memory, events, semaphores, message queues, blackboards,…
- Processes created at initialization only
- Real time operating system (ARINC 653) with fixed priorities (highest priority runs first)
- Scheduled on a single processor

Verified properties

- Absence of runtime errors
- Absence of unprotected data races

Semantics

- No memory consistency model for C
- Optimizing compilers consider sequential processes out of their execution context

```
            init:  flag1 = flag2 = 0
        process 1:          |         process 2:
flag1 = 1;                  | flag2 = 1;
if (!flag2)                 | if (!flag1)
{                           | {
  /* critical section */    |   /* critical section */
```

write to `flag1/2` and read of `flag2/1` are independent so can be reordered → error!

- We assume:
  - sequential consistency in absence of data race
  - for data races, values are limited by possible interleavings between synchronization points

Abstractions

- Based on Astrée for the sequential processes
- Takes scheduling into account
- OS entry points (semaphores, logbooks, sampling and queuing ports, buffers, blackboards, …) are all stubbed (using Astrée stubbing directives)
- Interference between processes: flow-insensitive abstraction of the writes to shared memory and inter-process communications

Example of application: FWS

- Degraded mode (5 processes, 100 000 LOCS):
  - 1h40 on 64-bit 2.66 GHz Intel server
  - 98 alarms
- Full mode (15 processes, 1 600 000 LOCS):
  - 50 h
  - 12 000 alarms !!! more work is being done !!! (e.g. analysis of complex data structures, logs, etc)

Conclusion

Cost-effective verification

- The rumor has it that:
  - Manuel validation (testing) is costly, unsafe, not a verification!
  - Formal proofs by theorem provers are extremely laborious and not reusable hence costly
  - Model-checkers do not scale up
- Why not try abstract interpretation?
  - Domain-specific static analysis scales and *can* deliver no false alarm (but this requires developments of the analyzer by specialists)

The End