

# Verified squared: does critical software deserve verified tools?

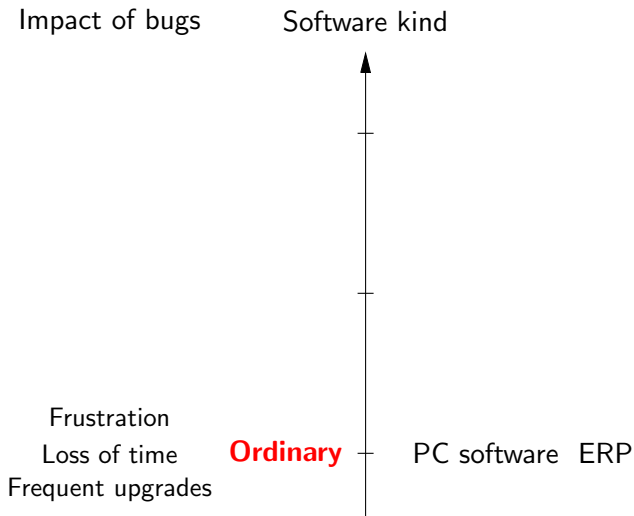
Xavier Leroy

INRIA Paris-Rocquencourt

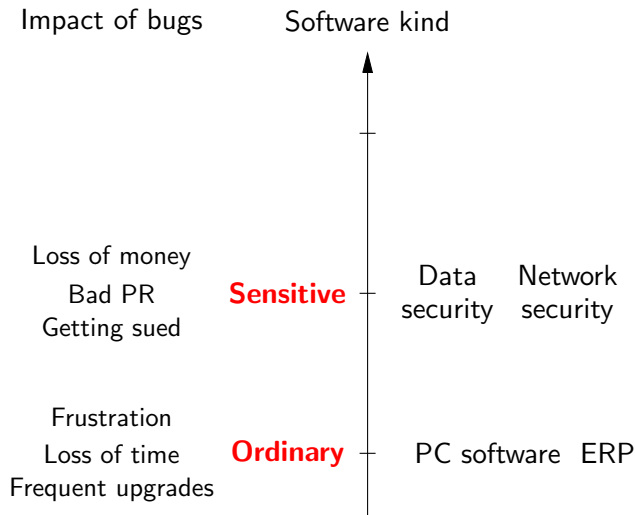
POPL 2011



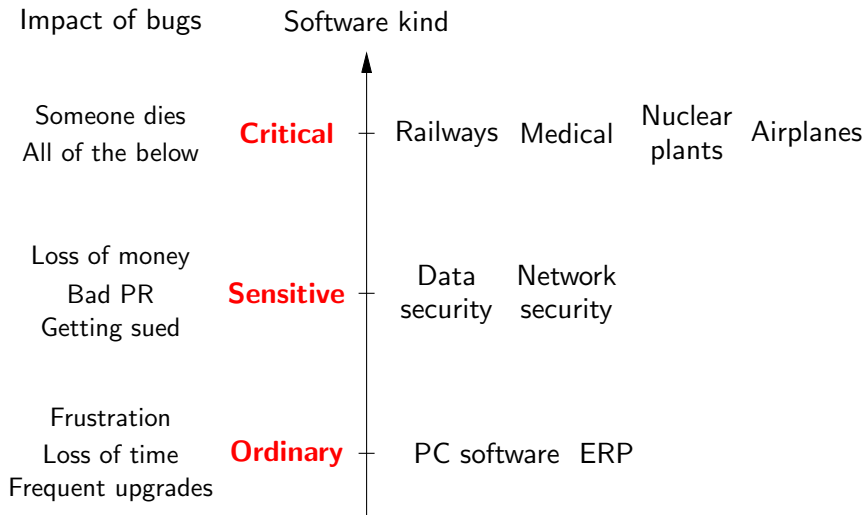
# The software reliability landscape



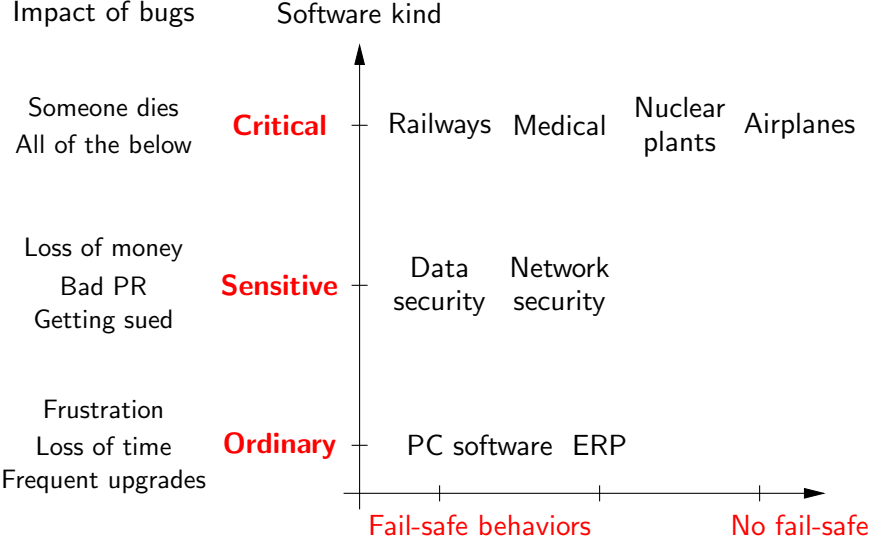
# The software reliability landscape



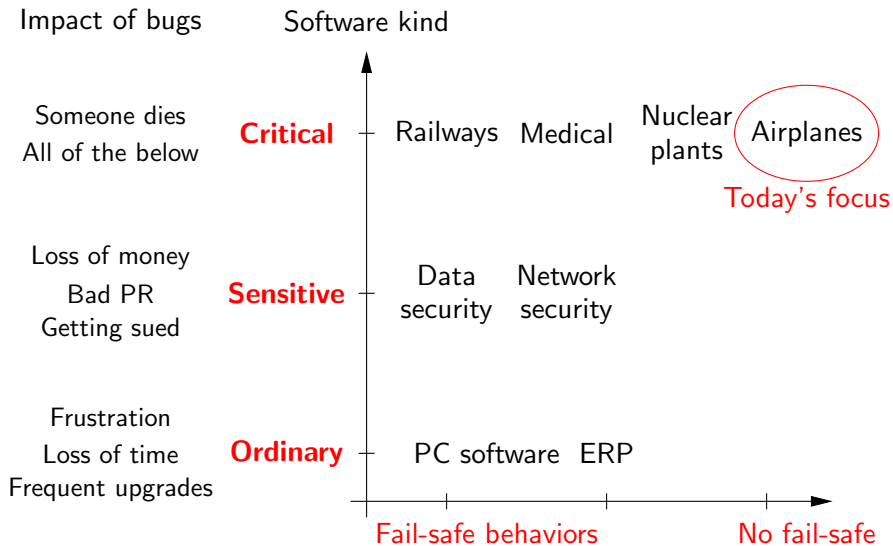
# The software reliability landscape



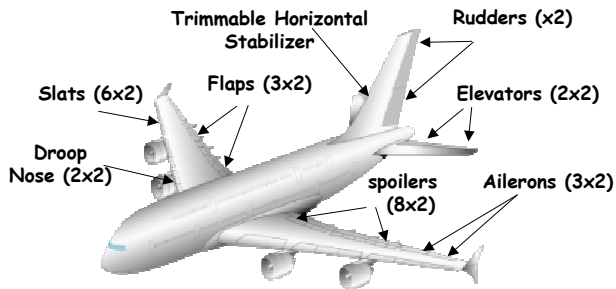
# The software reliability landscape



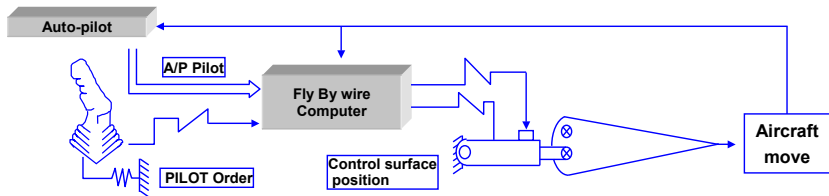
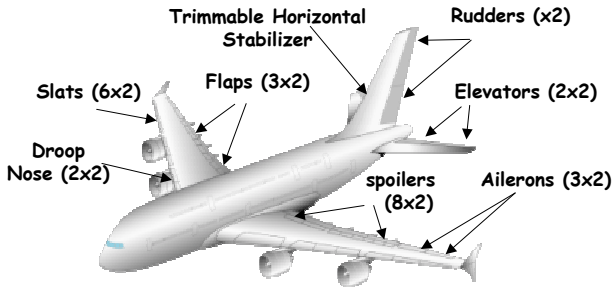
# The software reliability landscape



# Running example: fly-by-wire software



# Running example: fly-by-wire software



(G. Ladier)



# Timeline

1958

Avro CF 105  
(analog)

1969

Concorde  
(analog)

1984

Airbus 320  
(digital)

1995

Boeing 777  
(digital)

## EQUIPEMENTS AVIATION

### LA COMPLEXITE DES LOGICIELS EMBARQUES MENACE LA SECURITE DES AVIONS

Nombre d'incidents survenus récemment à des Boeing 747 sont imputés à des erreurs logicielles. Des problèmes analogues dans la mise au point des C-17 et F-14D restent une nouvelle fois en question si rien n'est attribué aux calculateurs embarqués.



Boeing 747

PHOTO: AIRBUS

3000 avions. Représentent pour un poids, le total des logiciels embarqués dans ces avions est évalué à 1000 mégaoctets. Le message est clair: la complexité des logiciels embarqués est en train de devenir un problème majeur pour les constructeurs d'avions.

## RESEARCH PAPER

### CAD: COMPUTER-AIDED DISASTER

PETER MELLOR  
Centre for Software Reliability,  
Northwood Research, London EC1A  
e-mail: p.mellor@cs.rsl.ac.uk

Computers can kill (or have other undesirable effects). This paper describes a number of recent disasters in which computers have been wholly or partly to blame, including the Therac-25, which caused hundreds of radiation deaths in patients, and the crashes of the British Airways and Air France Boeing 747s.

The great disk initialization disaster in 1986 a consultant had visited a known human rights organization computer system in its head office had just been delivered, and the accounting package had been on disk by the software house who had been asked to format a diskette.

## NEWS

COMPUTER WEEKLY AUGUST 13 1992

### Boeing opposes tests on safety-critical software

**Tony Collins**, Boeing aircraft manufacturer, says Boeing wants to dilute an already weakened FAA standard on the testing of safety-critical and other software.

In a letter to a standards committee it urged

come into operation next year. But Boeing says the checks would "go well beyond" the existing US Federal Aviation Regulations which cover software inspections and tests. The manufacturer says it is "opposed to the use of

validator could contest a rejection of its software. Also, Boeing says spot checks by certification authorities would undermine the tests carried by its employees. Validation of the life-cycle process should be left to the applicant (Boeing) rather than the certification authority.

Advisory Concepts for Aviation in Seattle, U.S., has already rejected calls from the British Computer Society for the mandatory independent certification of all safety-critical software used in computerized aircraft such as the Airbus A320. However, the committee has included a provision for

creating and revising liability and assigning the responsibility for all types of software. The committee is a type of safety facilities that it can sometimes be used. However,

# Functions of FBW software

High AOA Protection	Load Factor Limitation	Pitch Attitude Protection
<b>NORMAL LAW</b>		
High Speed Protection	Flight Augmentation (Yaw)	Bank Angle Protection

Execute pilot's commands.

Active damping of oscillations.

Flight assistance: keep aircraft within safe flight envelope.

Low Speed Stability	Load Factor Limitation	
<b>ALTERNATE LAW</b>		
High Speed Stability	Yaw Damping Only	

	Load Factor Limitation	
<b>ABNORMAL ALTERNATE LAW w/o Speed Stability</b>		
	Yaw Damping Only	

<b>DIRECT LAW</b>		

# Anatomy of FBW systems

Two-part software:

- A minimalistic operating system (C)  
(Boot, self-tests, communications over buses, static scheduling of periodic tasks. Generally hand-crafted, sometimes off-the-shelf.)
- Mostly: **control-command code**. (Scade)

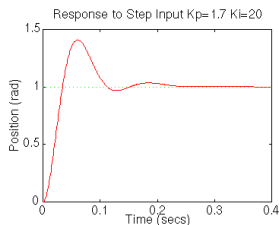
Hard real-time.

100k – 1M LOC of C code, but mostly generated from block diagrams.

Asymmetric redundancy (e.g. 3 primary units, 3 secondary).

# Control-command laws

“Hello world” example: the PID controller.



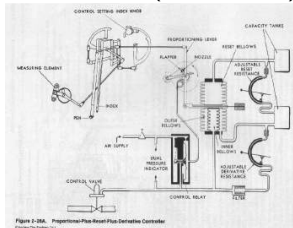
Error  $e(t) = \text{desired position}(t) - \text{current position}(t)$ .

$$\text{Action } a(t) = K_p e(t) + K_i \int_0^t e(t) dt + K_d \frac{d}{dt} e(t)$$

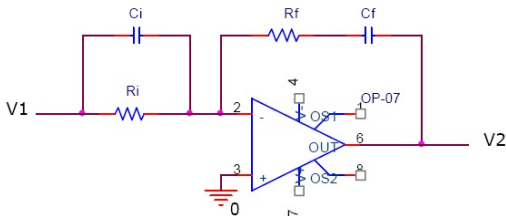
(Proportional)      (Integral)      (Derivative)

# Implementing a control law

Mechanical (pneumatic):



Analog electronics:

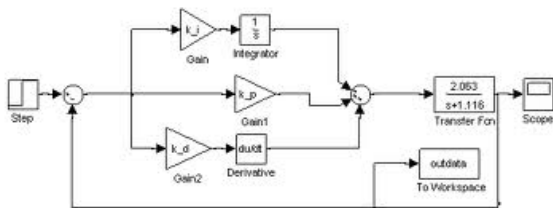


Software (the current favorite):

```
previous_error = 0; integral = 0
loop forever:
  error = setpoint - actual_position
  integral = integral + (error*dt)
  derivative = (error - previous_error)/dt
  output = (Kp*error) + (Ki*integral) + (Kd*derivative)
  previous_error = error
  wait(dt)
```

## Block diagrams (Simulink, Scade)

Such code is rarely hand-written, more often generated from diagrams:



For Scade, a graphical notation for the Lustre reactive language:

```
error = setpoint - position
integral = (0 -> pre(integral)) + error * dt
derivative = (error - (0 -> pre(error))) / dt
output = Kp * error + Ki * integral + Kd * derivative
```

defining **time-indexed sequences** via equations.

# Lustre: a successful domain-specific language

LUSTRE: A declarative language  
for programming synchronous systems\*

(POPL 1987.)

P. Caspi   D. Pilaud   N. Halbwachs   J. A. Plaice  
Laboratoire "Circuits et Systèmes"   Laboratoire de Génie Informatique  
BP68, 38402 St Martin d'Hères, FRANCE

Received 10/15/86

## Abstract

LUSTRE is a synchronous data-flow language for programming systems which interact with their environments in real-time. After an informal presentation of the language, we describe its semantics by means of structural inference rules. Moreover, we show how to use this semantics in order to generate efficient sequential code, namely, a finite state automaton which represents the control of the program. Formal rules for program transformation are also presented.

- Both in systems of equations and in operator nets, there is neither the notion of control nor that of sequentiality. The only constraints on the evaluation order arise from the dependencies between variables. As a consequence, any implementation, be it sequential or highly parallel, can be easily derived.
- As pointed out above, the considered equations are generally time invariant: variables may be considered to be functions of time, and  $X=E$  means that at each instant  $t$ ,  $x_t = e_t$ . Hence, such models are likely to

- Matches engineering practice (incl. visual syntax).
- Clean semantics.
- Low expressiveness: a language of boxes, wires, latches, and clocks.
- High potential for verification (model-checking, and more) and supercompilation (automata-based, and more).

# The qualification process (DO-178)



Rigorous validation:

- Review (qualitative)
- Analysis (quantitative)
- Testing (huge amounts, from unit tests to flying the plane)

Conducted at multiple levels, from designs to final product.

Meticulous development process; extensive documentation.



# Then, a miracle happens:

*The aircraft industry shows practical, economic interest in formal, tool-assisted verification of software!*

(Not so much for stronger guarantees, but primarily to save on testing.)

# Some success stories in verification of avionics code

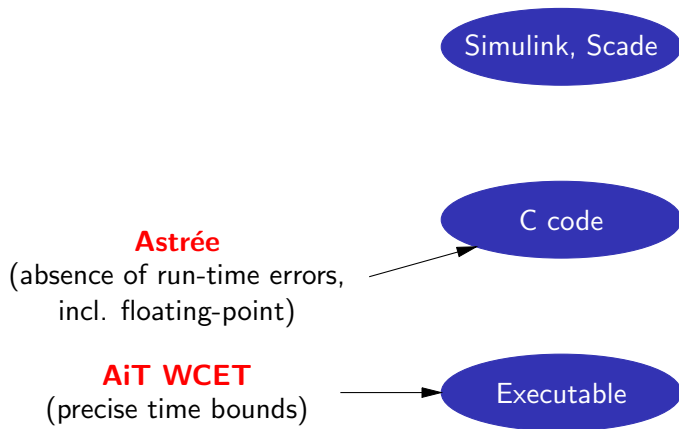
Simulink, Scade

C code

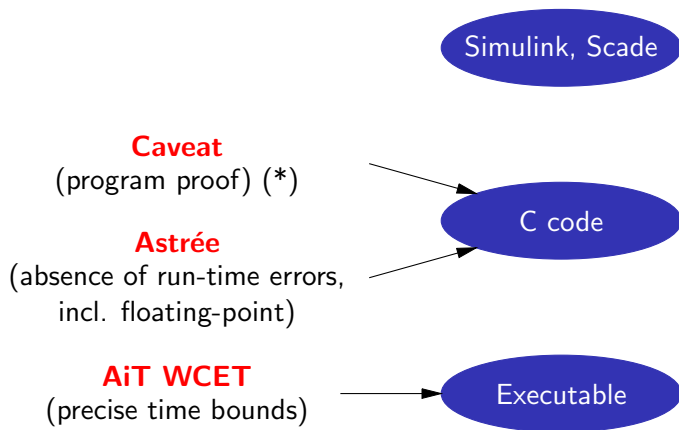
**AiT WCET**  
(precise time bounds)

Executable

# Some success stories in verification of avionics code



# Some success stories in verification of avionics code



(\*) Motto: "unit proofs as a replacement for unit tests"

# Some success stories in verification of avionics code

**Rockwell-Collins toolchain**

(model-checking + proof)



**Caveat**

(program proof) (\*)

**Astrée**

(absence of run-time errors,  
incl. floating-point)



**AiT WCET**

(precise time bounds)



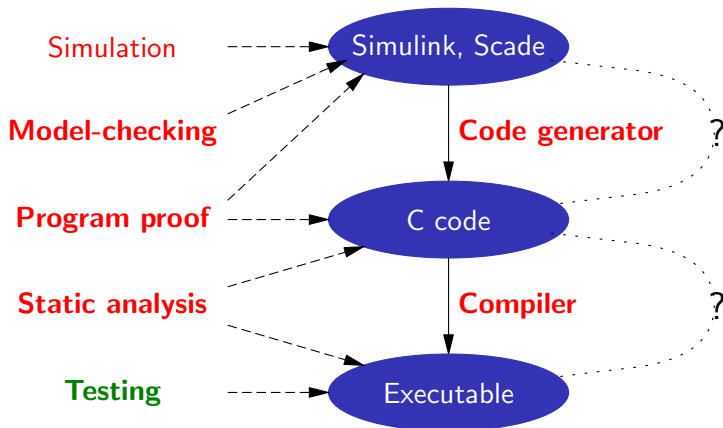
(\*) Motto: “unit proofs as a replacement for unit tests”

# Round of applause!

(For putting into practice Hoare logics and abstract interpretation,  
long considered as purely academic exercises)

But...

# Trust in formal verification



Are verification tools semantically sound?

Are compilers and code gen. semantics-preserving?

# Can you trust your compiler?



## Miscompilation happens

*NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.*

<http://www.nullstone.com/htmls/category/divide.htm>

*We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.*

*E. Eide & J. Regehr, EMSOFT 2008*

*We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 290 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.*

*J. Regehr et al, 2010*

# The DO-178B take on the issue

For code generators:

- Either the generator is qualified at the same level of assurance as the application itself  
(strict coding rules; meticulous dev. process; extensive testing)
- Or the generated code must be qualified as if hand-written  
(manual reviews, traceability, even more testing).

For compilers, neither option is realistic, hence various compromises:

- Manual review of generated asm on small representative codes.
- Naive, unoptimized compilation.

# The semanticist's take on the issue

Why not **formally verify the compiler itself?**

After all, compilers have simple specifications:

*If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.*

An idea as old as this speaker. . .

“Old pots make tasty soups” (French proverb)

John McCarthy  
James Painter<sup>1</sup>

## CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS<sup>2</sup>

**1. Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

*Mathematical Aspects of Computer Science, 1967*

# “Plus ça change, . . .” (American saying)

3

## Proving Compiler Correctness in a Mechanized Logic

---

R. Milner and R. Weyhrauch

Computer Science Department  
Stanford University

### Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

*Machine Intelligence (7), 1972.*

## APPENDIX 2: command sequence for McCarthy-Painter lemma

```

GOAL  $\forall e, sp, lswfse, e :: \text{MT}(\text{compe } e, sp) \Rightarrow \text{svof}(sp) \mid ((\text{MSE}(e, \text{svof } sp)) \& \text{pdof}(sp)),$ 
 $\forall e, lswfse, e :: \text{swft}(\text{compe } e) \Rightarrow \text{TT},$ 
 $\forall e, lswfse, e :: (\text{count}(\text{compe } e) = 0) \Rightarrow \text{TT};$ 

TRY 2 INDUCT 56;
TRY 1 SIMPL;
LABEL INDHYP;
TRY 2 ABSTR;
TRY 1 CASES wfs_ofun(f,e);
LABEL TT;
TRY 1 CASES type a="N";
TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISHFT1,,FCOUNT;
TRY 2;SS-,TT;SIMPL,TT;QED;
TRY 3 CASES type a="E";
TRY 1 SUBST ,FCOMPE;
SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
INCL-,1;SS+--;INCL--,2;SS+--;INCL---,3;SS+--;
TRY 1 CONJ;
TRY 1 SIMPL;
TRY 1 USE COUNT1;
TRY 1;
APPL ,INDHYP+2, a, pdof e;
LABEL CARG1;
SIMPL-;QED;
TRY 2 USE COUNT1;
TRY 1;

```

(Even the proof scripts look somewhat familiar)

# The CompCert project

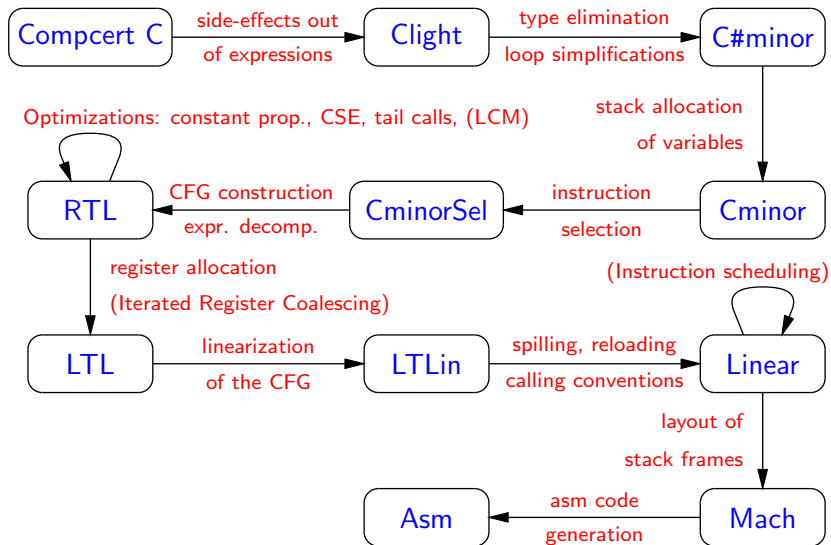
(X.Leroy, S.Blazy, et al)

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code  
⇒ careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.

# The formally verified part of the compiler





# Formally verified in Coq

After 50 000 lines of Coq and 4 person.years of effort:

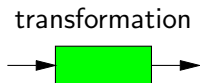
Theorem `transf_c_program_is_refinement`:

```
forall p tp,  
transf_c_program p = OK tp ->  
(forall beh, exec_C_program p beh -> not_wrong beh) ->  
(forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

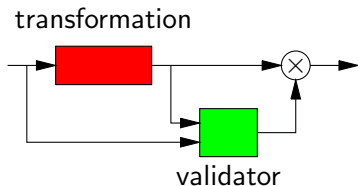
Behaviors `beh` = termination / divergence / going wrong  
+ trace of I/O operations (syscalls, volatile accesses).

# Compiler verification patterns (for each pass)

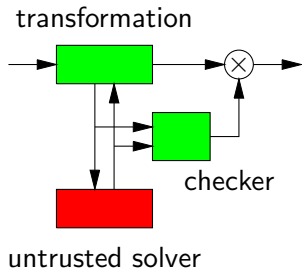
## Verified transformation




## Verified translation validation



## External solver with verified validation



 = formally verified

 = not verified

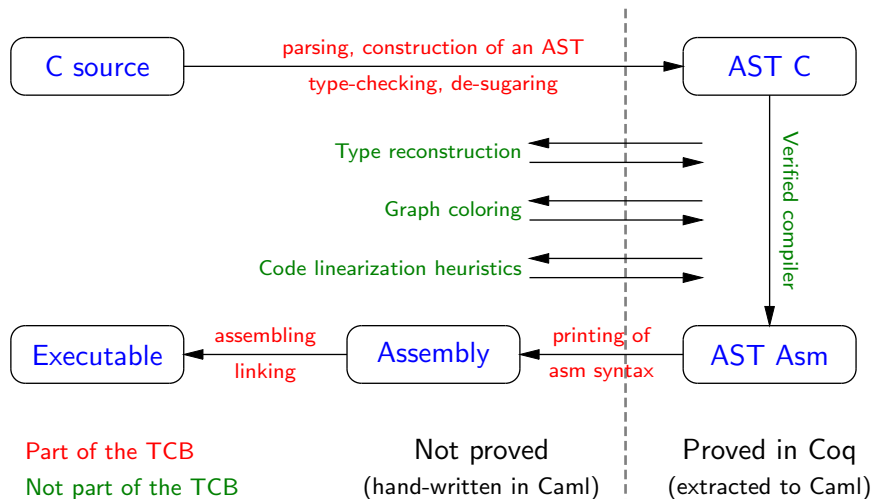
# Programmed in Coq

Compilation algorithms written in Coq's specification language, in pure functional style.

```
Fixpoint transl_expr (map: mapping) (a: expr) (rd: reg) (nd: node)
  {struct a}: mon node :=
  match a with
  | Evar v =>
    do r <- find_var map v; add_move r rd nd
  | Eop op al =>
    do rl <- alloc_regs map al;
    do no <- add_instr (Iop op rl rd nd);
    transl_exprlist map al rl no
  | Eload chunk addr al =>
    do rl <- alloc_regs map al;
    do no <- add_instr (Iload chunk addr rl rd nd);
    transl_exprlist map al rl no
  | ...
```

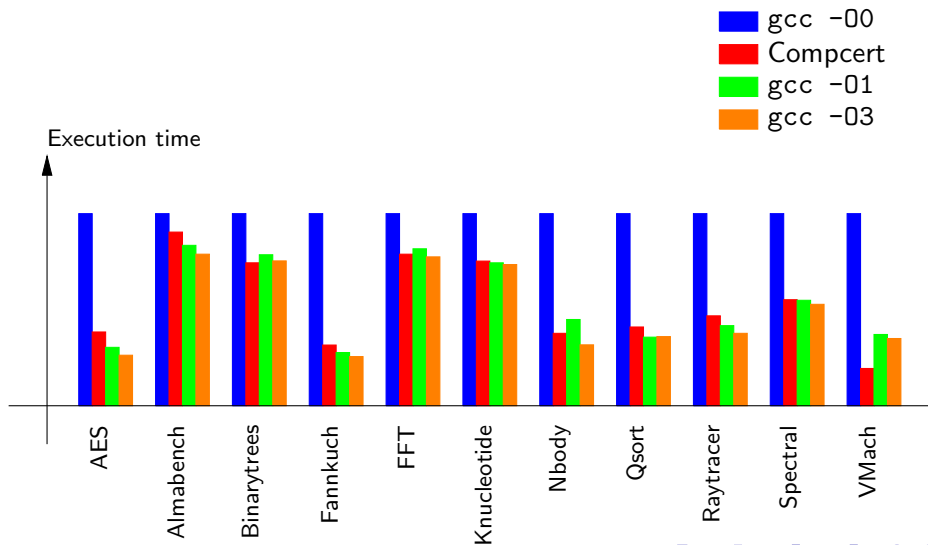
Executable via automatic extraction to Caml.

# The whole Compcert compiler



# Performance of generated code

(On a PowerPC G5 processor)



## Current status

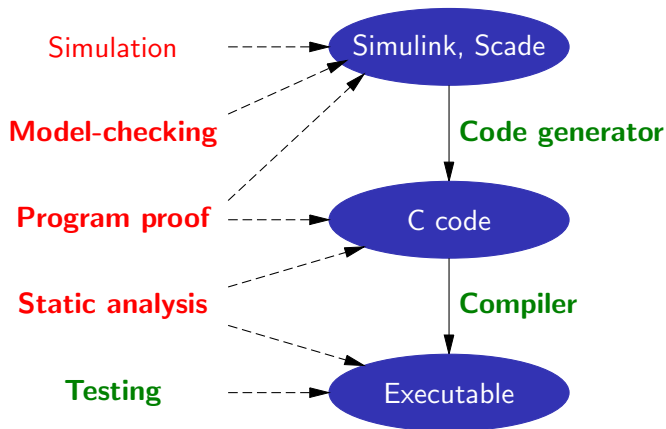
The formal verification of realistic compilers is feasible.  
(Within the limitations of contemporary proof assistants.)

Much work remains:

- Shrinking the TCB  
(e.g. verified parsing, validated assembling & linking).
- More optimizations.
- Front-ends for other languages.  
(see T. Ramananandro's talk for elements of C++).
- Concurrency!  
(see J. Sevcik's talk & A. Appel et al's *Verified Software Toolchain*).
- Connections with source-level verification (next).

# Can you trust your verification tools?

# Trust in formal verification (again)





# Requirements on verification tools

(Static analyzers, program provers, model-checkers)

**When used as sophisticated bug-finders:** everything goes.  
(Unsound static analyzers have their uses, e.g. Coverity.)

**When used to establish properties of programs** and remove the corresponding tests: evidence of **soundness** is required.

DO-178-B: lightweight qualification for verification tools, as they cannot introduce bugs, just miss the existence of bugs. But why stop here?

# A holistic effect with compiler verification

## Stronger correctness results:

```
forall p tp,  
transf_c_program p = OK tp ->  
(forall beh, exec_C_program p beh -> not_wrong beh) ->  
(forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

## Simpler, more precise verification tools:

they know exactly how the compiler implements unspecified behaviors of C.

# A holistic effect with compiler verification

## Stronger correctness results:

```
forall p tp,  
transf_c_program p = OK tp -> Astree_result p = true ->  
(forall beh, exec_C_program p beh -> not_wrong beh) ->  
(forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

## Simpler, more precise verification tools:

they know exactly how the compiler implements unspecified behaviors of C.

# Example: static analysis by abstract interpretation

The orthodox presentation:

A collecting semantics  $\quad + \quad$  A Galois connection  $(\mathcal{P}(S), \subseteq) \xrightleftharpoons[\alpha]{\gamma} (A, \sqsubseteq)$

From there, abstract operations can be **calculated** in a correct-by-construction way.

5 — When  $A = A_1 \text{ b } A_2$  is a binary operation, we have

$$\begin{aligned} & \alpha^*(\text{Faexp}[A_1 \text{ b } A_2])r \\ = & \alpha(\{v \mid \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \text{ b } A_2 \Rightarrow v\}) \\ = & \{ \text{def. (27) of } \rho \vdash A_1 \text{ b } A_2 \Rightarrow v \} \\ \sqsubseteq & \alpha(\{v_1 \text{ b } v_2 \mid \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \Rightarrow v_1 \wedge \rho \vdash A_2 \Rightarrow v_2\}) \\ \sqsubseteq & \{ \alpha \text{ monotone (5)} \} \\ \sqsubseteq & \alpha(\{v_1 \text{ b } v_2 \mid \exists \rho_1 \in \dot{\gamma}(r) : \rho_1 \vdash A_1 \Rightarrow v_1 \wedge \exists \rho_2 \in \dot{\gamma}(r) : \rho_2 \vdash A_2 \Rightarrow v_2\}) \\ \sqsubseteq & \{ \gamma \circ \alpha \text{ is extensive (6), } \alpha \text{ is monotone (5)} \} \\ \sqsubseteq & \alpha(\{v_1 \text{ b } v_2 \mid v_1 \in \gamma \circ \alpha(\{v \mid \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_1 \Rightarrow v\}) \wedge \\ & \quad v_2 \in \gamma \circ \alpha(\{v \mid \exists \rho \in \dot{\gamma}(r) : \rho \vdash A_2 \Rightarrow v\})\}) \\ \sqsubseteq & \{ \text{induction hypothesis (33), } \gamma \text{ (4) and } \alpha \text{ (5) are monotone} \} \\ \sqsubseteq & \alpha(\{v_1 \text{ b } v_2 \mid v_1 \in \gamma(\text{Faexp}[A_1]r) \wedge v_2 \in \gamma(\text{Faexp}[A_2]r)\}) \\ \sqsubseteq & \{ \text{by defining } \mathbf{b}^* \text{ such that } \mathbf{b}^*(p_1, p_2) \sqsupseteq \alpha(\{v_1 \text{ b } v_2 \mid v_1 \in \gamma(p_1) \wedge v_2 \in \gamma(p_2)\}) \} \\ = & \mathbf{b}^*(\text{Faexp}[A_1]r, \text{Faexp}[A_2]r) \\ = & \{ \text{by defining } \text{Faexp}^*[A_1 \text{ b } A_2]r \triangleq \mathbf{b}^*(\text{Faexp}[A_1]r, \text{Faexp}[A_2]r) \} \\ & \text{Faexp}^*[A_1 \text{ b } A_2]r . \end{aligned}$$

(P. Cousot)

# Towards a Coq mechanization

(D. Pichardie, 2005, 2008; D. Cachera and D. Pichardie, 2010)

Problem:  $\alpha$  functions not computable; poor support for calculational style.

Solution 1: forget about  $\alpha$ ; use relations:

$$\vdash \text{concrete-thing} \in \text{abstract-thing}$$

Solution 2: use calculations on paper and re-prove resulting definitions:

$$\vdash v_1 \in A_1 \wedge \vdash v_2 \in A_2 \implies \vdash v_1 + v_2 \in A_1 +^{\#} A_2$$

Much heroic Coq work remains:

- Many, many abstract operations to prove.
- Modular construction of abstract domains.
- Fixpoints, with widening and narrowing.
- Their termination.

# Cutting more corners

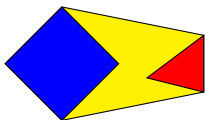
(D. Pichardie et al, 2010)

Forget about mechanically proving termination.

(Coq + classical logic can reason over partial functions.)

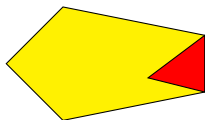
Validate *a posteriori* a post-fixpoint computed by untrusted code.

(Also reduces the number of abstract operations to prove.)



Computing joins  
(convex hull)

vs.



Checking inclusion  
(Presburger formula)

# Summary

Medium term: good hope for a verified, non-toy static analyzer.

( $\approx 1/10$  of Astrée, just like CompCert  $\approx 1/10$  of gcc.)

Similar work in progress on verified program provers.

(E.g. A. Appel et al's *Verified Software Toolchain*.)

A vision: what you compile is *exactly* what you verified.

*This is a public service announcement:*

# Floating-point needs work!



# Floating-point arithmetic

A. S. Householder (1904–1993):

*“It makes me nervous to fly an airplane since I know they are designed using floating-point arithmetic.”*

# Floating-point arithmetic

A. S. Householder (1904–1993):

*“It makes me nervous to fly an airplane since I know they are designed using floating-point arithmetic.”*

X. Leroy (1968–) and many, many others:

*“It makes us nervous to fly an airplane since we know they **OPERATE** using floating-point arithmetic.”*

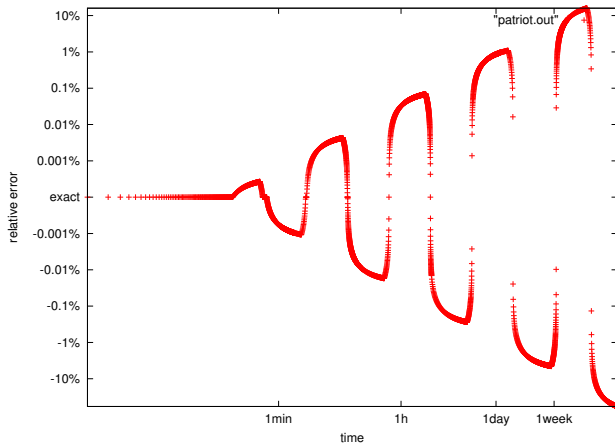
# Floating-point 101

$$m \cdot 2^e \quad \text{with } |m| < 1$$

- **Limited precision** for  $m$  (e.g. 23 or 52 digits)  $\Rightarrow$  inaccuracies  
(Patriot missile failure: 28 casualties)
- **Limited range** for  $e$   $\Rightarrow$  overflow, underflow  
(Ariane 5 maiden flight: \$500M)
- **Special values:**  $+\infty$ ,  $-\infty$ ,  $+0.0$ ,  $-0.0$ , Not-a-Number.

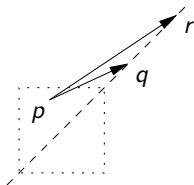
# Accumulated rounding errors (a.k.a. the Patriot bug)

```
float t = 0.0; while(1) { ... t = t + 0.1; ... }
```

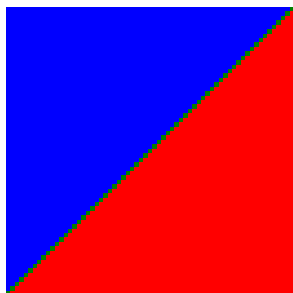


# Catastrophic cancellation

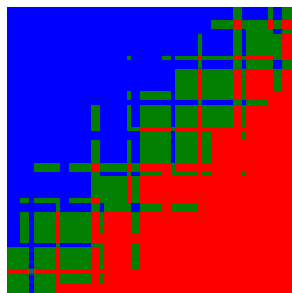
```
float det = (qx - px) * (ry - py)
           - (qy - py) * (rx - px);
if (det > 0) return RIGHT;
if (det < 0) return LEFT;
return ALIGNED;
```



- aligned
- oriented ↻
- oriented ↻



Expected



Actual

(G. Melquiond)

# Unfounded compiler optimizations

A post-it note for compiler writers: (please stick it to your monitor)

✗ *Thou shalt not assume...*

$$x == x$$

$$x \leq y \Leftrightarrow \neg(x > y)$$

$$x == y \Rightarrow 1/x == 1/y$$

$$x / 10 == x * 0.1$$

$$x + (y + z) == (x + y) + z$$

$$x * (y * z) == (x * y) * z$$

$$\text{rnd}_{64}(\text{rnd}_{80}(op)) == \text{rnd}_{64}(op)$$

✓ *although you can assume...*

$$x == y \Leftrightarrow x - y == 0$$

$$x \leq y \Leftrightarrow x < y \vee x == y$$

$$x == y \Rightarrow 1+x == 1+y$$

$$x / 8 == x * 0.125$$

$$x + y == y + x$$

$$x * y == y * x$$

$$\text{rnd}_{32}(\text{rnd}_{64}(op)) == \text{rnd}_{32}(op)$$

## Wishy-washy language definitions

E.g. ISO C99:

*6.3.1.8[2]: The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.*

Gives the C compiler a lot of rope to hang the programmer with.

## Wishy-washy language definitions

E.g. ISO C99:

*6.3.1.8[2]: The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.*

Gives the C compiler a lot of rope to hang the programmer with.

Witness GCC bug #323 “optimized code gives strange floating point results”: reported in 2000, dozens of duplicates, 179 comments, still not acknowledged. . .

. . . and responsible for PHP’s `strtod()` function not terminating (→ denial of service on many Web sites).



# Time for sledgehammer theorems

## Theorem (Boldo-Nguyen)

Let  $\odot$  be an operation among addition, subtraction, multiplication, division, square root, negation and absolute value.

Let  $\square$  be the rounding mode chosen by the compiler (among 64, 80, 80-then-64, or exact).

Let  $x = \odot(y, z)$  be the exact result of the operation.

Then, the computed result  $\square(x)$  is such that

- If  $|x| \geq 2^{-1022}$  then  $\square(x) \in [x \pm 2050 \cdot 2^{-64} \cdot |x|] \setminus ]-2^{-1022}, 2^{-1022}[$
- If  $|x| < 2^{-1022}$  then  $\square(x) \in [x \pm 2049 \cdot 2^{-1086}] \cap [-2^{-1022}, 2^{-1022}]$

# Verification of floating-point code

## By static analysis:

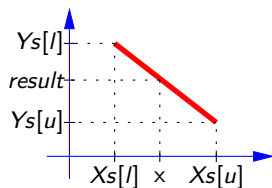
- Non-relational (intervals): not too hard (widen intervals conservatively).
- Relational (polyhedra, octogons): very delicate (mechanized proofs would be highly welcome).
- Specialized (Fluctuat): scaling issues.

## By program proof:

- Difficult proofs, even with the help of dedicated decision procedures (e.g. G. Melquiond's GAPPa).
- Specifications are unclear to begin with!

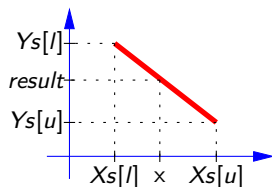
## Proving a typical Scade symbol: tabulated function

```
double tabulate(int N, double Xs[N+1], double Ys[N+1], double x)
{
  int l = 0, u = N;
  while (l + 1 < u) {
    int m = (l + u) / 2;
    if (x < Xs[m]) u = m; else l = m;
  }
  double p = (x - Xs[l]) / (Xs[u] - Xs[l]);
  return (1 - p) * Ys[l] + p * Ys[u];
}
```



## Proving a typical Scade symbol: tabulated function

```
double tabulate(int N, double Xs[N+1], double Ys[N+1], double x)
{
  int l = 0, u = N;
  while (l + 1 < u) {
    int m = (l + u) / 2;
    if (x < Xs[m]) u = m; else l = m;
  }
  double p = (x - Xs[l]) / (Xs[u] - Xs[l]);
  return (1 - p) * Ys[l] + p * Ys[u];
}
```

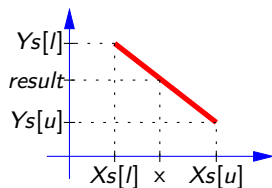


Binary search: business as usual

Pre:  $Xs[0] < \dots < Xs[N] \wedge Xs[0] \leq x < Xs[N] \wedge 1 \leq N < \text{MAX\_INT}/2$   
Post:  $\exists i, 0 \leq i < N \wedge Xs[i] \leq x < Xs[i + 1]$  ✓

## Proving a typical Scade symbol: tabulated function

```
double tabulate(int N, double Xs[N+1], double Ys[N+1], double x)
{
  int l = 0, u = N;
  while (l + 1 < u) {
    int m = (l + u) / 2;
    if (x < Xs[m]) u = m; else l = m;
  }
  double p = (x - Xs[l]) / (Xs[u] - Xs[l]);
  return (1 - p) * Ys[l] + p * Ys[u];
}
```



What can we say about the return value? Not much, really!

Post:  $\dots \wedge \min(Ys[i], Ys[i + 1]) \leq result \leq \max(Ys[i], Ys[i + 1])$  ✗

$Xs[i]$	$x$	$Xs[i + 1]$	$Ys[i]$	$Ys[i + 1]$	$p$	$result$
$-2^{1022}$	$2^{1022}$	$1.5 \cdot 2^{1022}$	-	-	NaN	NaN
0	0.5	1	$2^{-1074}$	$2^{-1074}$	0.5	0

In closing. . .

## In closing...

The formal verification of development and verification tools for critical software

... is a fascinating challenge,

... appears within reach,

... and could have practical importance.

*Feel free to join!*