In search of software perfection

Xavier Leroy

Inria Paris

Milner award lecture, Royal Society, 2016-11-24

Part I

Imperfect software

Software crashes. . .

Paris highway

Las Vegas billboard

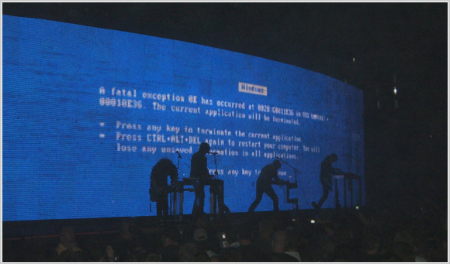
Software crashes. . .

Metro station, Manhattan

Heathrow airport

Software crashes. . .

Olympic games, 2008

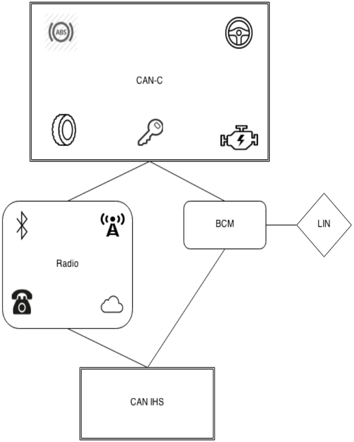


Nine Inch Nails concert

Software has security holes. . .

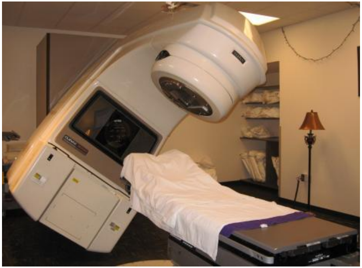
Attacker can remotely control many of the car's functions.

Fiat-Chrysler recalled 1.5 M vehicles for software update.

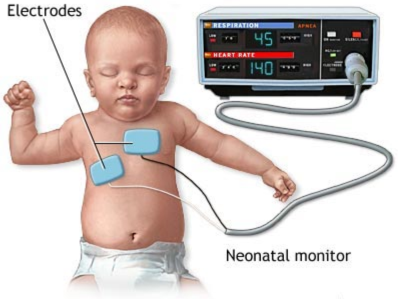


Remote Exploitation of an Unaltered Passenger Vehicle, C. Miller and C. Valasek, 2015

Software kills. . .



Therac 25 radiation machine
(3 patients dead following
massive overdose.)

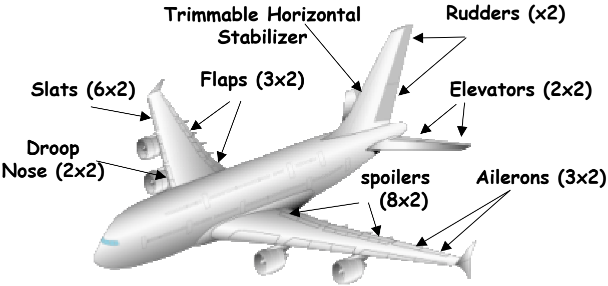


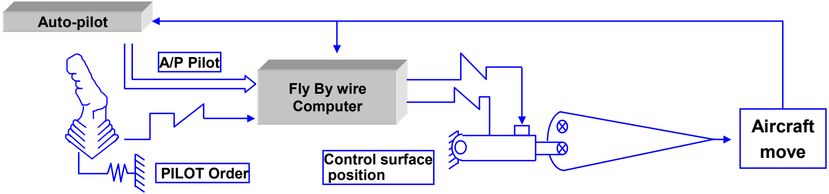
Newborn monitor
(several cases of sudden infant death
where the alarm did not ring)

Part II

A glimpse of hope: Critical avionics software

Running example: fly-by-wire software





(G. Ladier)

Timeline

1958	1969	1984	1995
Avro CF 105 (analog)	Concorde (analog)	Airbus 320 (digital)	Boeing 777 (digital)

ÉQUIPEMENTS AVIONIQUE

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 remettent une nouvelle fois en question le rôle qu'on attribue aux calculateurs embarqués.

British Airways signale 10 accidents d'origine logicielle survenus notamment sur ses 747.

PAR JEAN DUPONT

RESEARCH PAPER

CAD: COMPUTER-AIDED DISASTER

PETER MELLOR

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 ... overdoses of radiation to its patients, ...

The great disk initialization disaster

In 1986 a consultant had volunte... known human rights organizatio... computer system in its head office had just been delivered, and the ... accounting package had been ... disk by the software house whic...

NEWS

COMPUTER WEEKLY AUGUST 13 1992

Boeing opposes tests on safety-critical software

Tony Collins

US AIRCRAFT manufacturer Boeing wants to dilute an already weakened aviation standard on the testing of safety-critical and other software.

In a letter to a standards-setting committee it urges ...

... 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 ...

... turer 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 processes should be left to the applicant (Boeing) rather than the certification author-...

... nical Concepts for Aviation in Seattle, US, has already rejected calls from the British Computer Society for the mandatory independent testing of all safety-critical software used in computerised aircraft such as the Airbus A320.

However, the committee has included a provision in ...

Functions of FBW software

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

Low Speed Stability	Load Factor Limitation	
ALTERNATE LAW		
High Speed Stability	Yaw Damping Only	

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

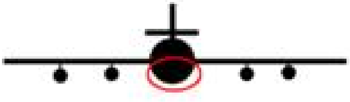
DIRECT LAW		

Execute pilot's commands.

Flight assistance: keep aircraft within safe flight envelope.

Fuel economy: minimize drag.

Active damping of oscillations.



Anatomy of FBW systems

Two-part software:

- A minimalistic operating system (written in C)
  (Boot, self-tests, communications over buses, static scheduling of periodic tasks. Generally hand-crafted, sometimes off-the-shelf.)
- Mostly: control-command code (in Simulink/Scade)
  ($\approx$ discretized differential equations)

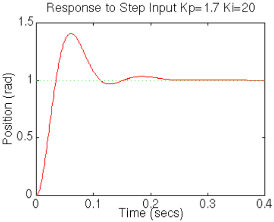
Hard real-time.

100k – 1M LOC of C code, mostly generated from Scade/Simulink.

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

Implementing a control law

"Hello, world" example: PID controller.



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

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 (e.g. pneumatic):

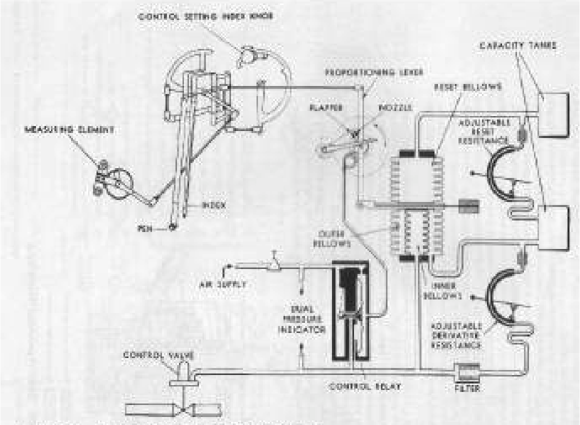
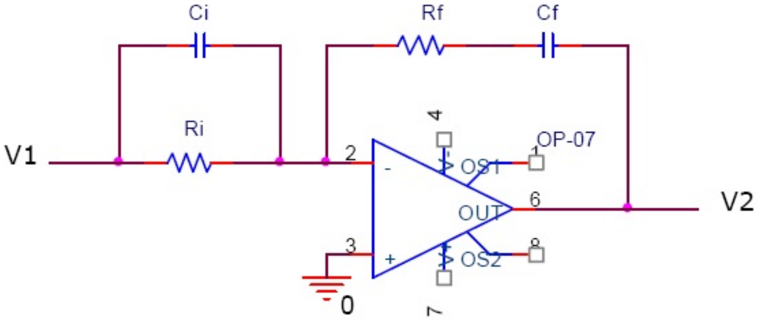


Figure 2-28A. Proportional-Plus-Reset-Plus-Derivative Controller

Implementing a control law

Analog electronics:



Implementing a control law

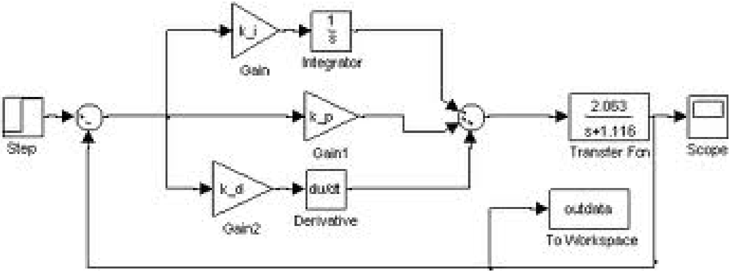
In software (today's favorite solution):

```
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)
```

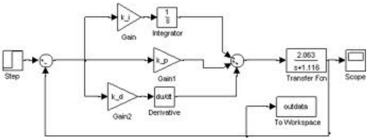
Block diagrams

(Simulink, Scade, Scicos, etc)

This kind of code is rarely hand-written, but rather auto-generated from block diagrams:



Block diagrams and reactive languages



In the case of Scade, this diagram is a graphical syntax 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
```

(= Time-indexed series defined by recursive equations.)

Block diagrams and reactive languages

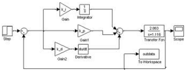
Control law

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

(modeling)

(discretization)

Block diagram



(syntax)

Recursive sequences

(semantics)

Lustre code

$$\begin{aligned} i_n &= i_{n-1} + e_n.dt \\ d_n &= (e_n - e_{n-1})/dt \\ o_n &= K_p e_n + K_i i_n + K_d d_n \end{aligned}$$

(code generation)

(hand-coding)

C code

Lustre: an example of a successful domain specific language.

The certification process (DO-178)

Design and development process is meticulous and fully documented.

Rigorous validation at multiple levels (from design to product):

- Reviews (qualitative)
- Analyses (quantitative)
- Test, test!, test!!, test, test, test, test, . . .
- Recent development: use of formal verification tools.

From unit testing. . .

```
double max(double x, double y)
{
    if (x >= y) return x; else return y;
}
```

```
max(0,0) = 0
max(0,1) = 1
max(0,-1) = 0
max(0,3.14) = 3.14
max(0,inf) = inf
max(0,-inf) = 0
max(1,0) = 1
max(1,1) = 1
```

```
max(1,-1) = 1
max(1,3.14) = 3.14
max(1,inf) = inf
max(inf,0) = inf
max(inf,-inf) = inf
max(nan,0) = 0
max(0,nan) = nan
```

. . . to integration testing. . .

. . . to exploration on an Iron Bird. . .

. . . to test flights

Chronicle / Michael Macor

Part III

Tool-assisted formal verification

Beyond testing: formal verification

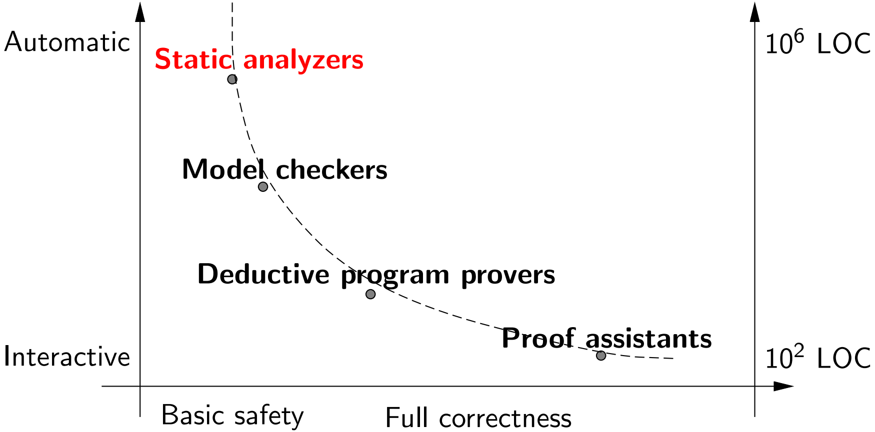
> *Program testing can be used to show the presence of bugs,*
> *but never to show their absence!*
>
> *(E.W.Dijkstra, 1972)*

Formal verification of software:
verify, possibly infer, properties that hold of all possible executions of a program.

Used in some industrial contexts (airplanes, railways)

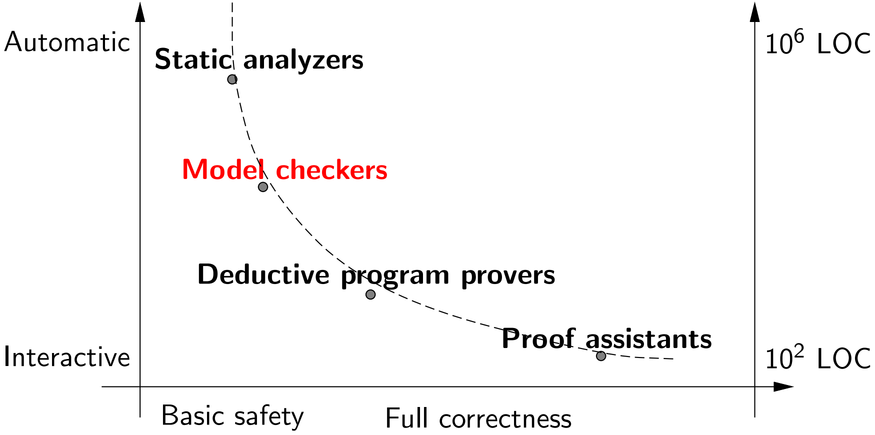
- To obtain independent guarantees (besides testing).
- To obtain stronger guarantees (than with testing).
- To replace costly unit tests.

A panorama of verification tools



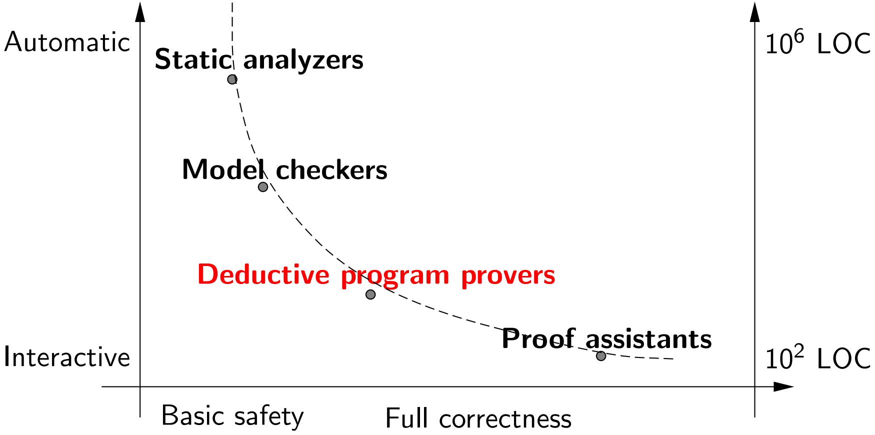
Static analysis: automatically infer simple properties of one variable ($x \in [N_1, N_2]$, $x \bmod N = 0$, etc) or several ($x + y \leq z$).

A panorama of verification tools



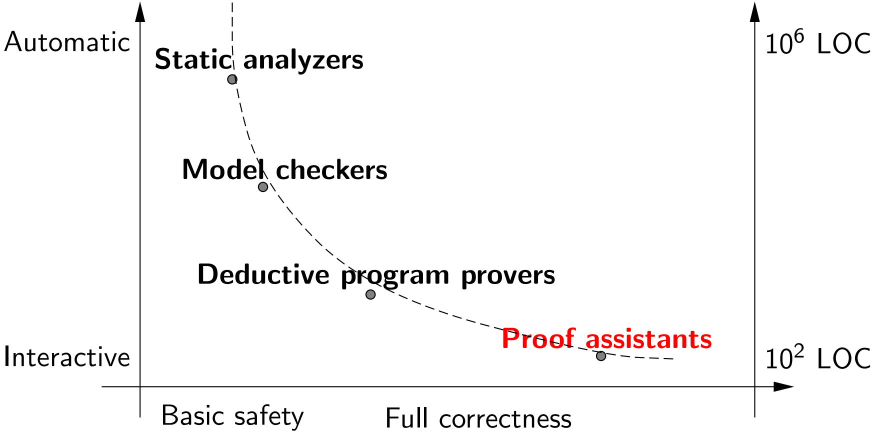
Model checking: automatically check that some "bad" program points are not reachable.

A panorama of verification tools



Program proof: show that
preconditions $\Rightarrow$ *invariants* $\Rightarrow$ *postconditions*
using automated theorem provers.

A panorama of verification tools



Proof assistants: conduct mathematical proofs in interaction with the user; re-check the proofs for correctness.

Example: computing prime numbers

```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
      int j = 0;
      while (j < i ∧ a[j] <= √m ) {
          if (a[j] divides m) continue loop;
          j = j + 1;
      }
      a[i] = m; i = i + 1;
  }
```

Goal: compute the first n prime numbers.

Algorithm: try successive odd numbers m, striking out those divisible by primes already found.

Example: computing prime numbers

```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
      int j = 0;
      while (j < i ∧ a[j] <= √m ) {
          if (a[j] divides m) continue loop;
          j = j + 1;
      }
      a[i] = m; i = i + 1;
  }
```

Static analyzer: can infer $1 \leq i < n$ and $0 \leq j < i$ inside the loop, hence array accesses are safe (within bounds).

Example: computing prime numbers

```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
      int j = 0;
      while (j < i ∧ a[j] <= √m ) {
          if (a[j] divides m) continue loop;
          j = j + 1;
      }
      a[i] = m; i = i + 1;
  }
```

Automatic program prover: can prove partial correctness if the user provides detailed loop invariants and simple axioms about primality and divisibility. (Termination is harder to prove.)

Example: computing prime numbers

```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
    /* invariant:
```

$$\forall k,\ \ 0 \leq k < i \ \ \Rightarrow \ \ isprime(a[k])$$

$$\forall p,\ \ 2 \leq p < m \wedge isprime(p) \ \ \Rightarrow \ \ \exists k,\ \ 0 \leq k < i \wedge a[k] = p$$

$$\forall k, m,\ \ 0 \leq k < j < i \ \ \Rightarrow \ \ a[k] < a[j]$$

```
    */
```

Automatic program prover: can prove partial correctness if the user provides detailed loop invariants and simple axioms about primality and divisibility. (Termination is harder to prove.)

Example: computing prime numbers

Knuth, *The Art of Computer Programming*, vol.1

```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
      int j = 0;
      while (j < i ∧ a[j] <= √m ) {
          if (a[j] divides m) continue loop;
          j = j + 1;
      }
      ...
```

Knuth's cunning optimization: the test `j < i` is redundant and can be omitted. Can you see why? Because of Bertrand's postulate!

Theorem (Chebychev, 1850; Erdös, 1932; Coq proof: Théry, 2002)

For all $n > 1$, there exists a prime p in $]n, 2n[$.

Example: computing prime numbers

Knuth, *The Art of Computer Programming*, vol.1

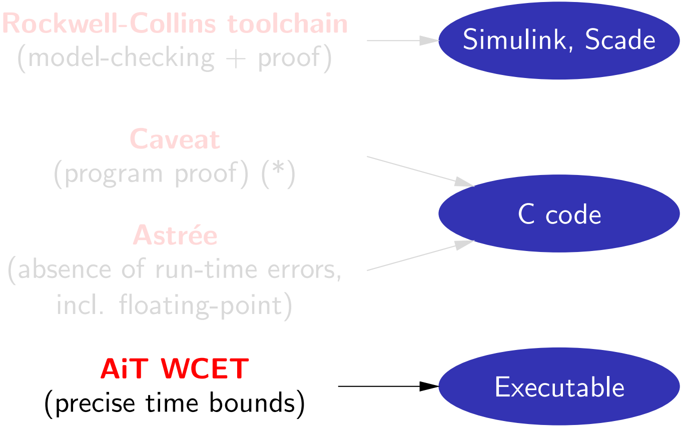
```
  int a[] = new int[n];
  a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
      int j = 0;
      while (~~j < i ∧~~ a[j] <= √m ) {
          if (a[j] divides m) continue loop;
          j = j + 1;
      }
      ...
```

Knuth's cunning optimization: the test `j < i` is redundant and can be omitted. Can you see why? Because of Bertrand's postulate!

Theorem (Chebychev, 1850; Erdös, 1932; Coq proof: Théry, 2002)

For all $n > 1$, there exists a prime p in $]n, 2n[$.

Success stories in verification of avionics code



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

Success stories in verification of avionics code

Rockwell-Collins toolchain
(model-checking + proof) → Simulink, Scade

Caveat
(program proof) (*) → C code

Astrée
(absence of run-time errors, incl. floating-point) → C code

AiT WCET
(precise time bounds) → Executable

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

Success stories in verification of avionics code

Rockwell-Collins toolchain
(model-checking + proof) → Simulink, Scade

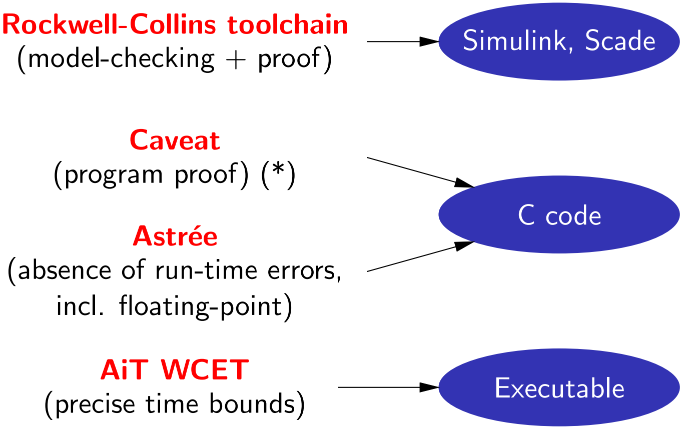
Caveat
(program proof) (*) → C code

Astrée
(absence of run-time errors, incl. floating-point) → C code

AiT WCET
(precise time bounds) → Executable

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

Success stories in verification of avionics code



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

Success stories in verification of systems code

The seL4 secure microkernel: (NICTA, 2009)

- Full correctness proof of a high-performance microkernel.
- Using the Isabelle/HOL proof assistant + custom automation.
- 8 KLOC of C code, 200 KLOC proof, 20 person.years.
- The largest deductive verification of a software system ever.

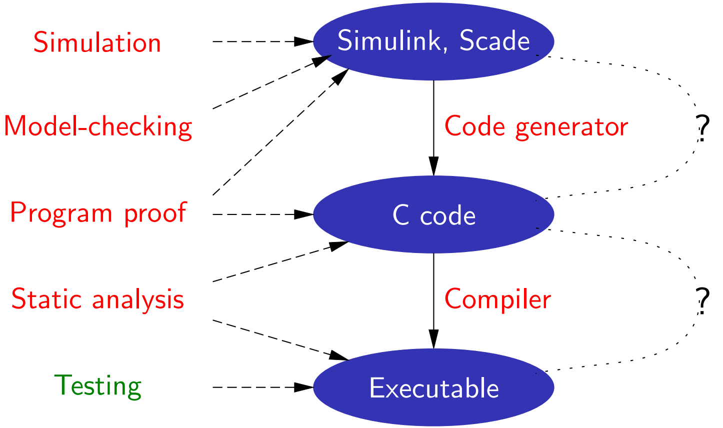
The Yxv6 file system: (U. Washington, 2016)

- Formally proved correct even in the presence of crashes.
- Automated verification using the custom Yggdrasil tool.

Part IV

Formally-verified compilation

Trust in software verification



The unsoundness risk: Are verification tools semantically sound?

The miscompilation risk: Are compilers semantics-preserving?

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

To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.

X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011

An example of optimizing compilation

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{i<n} a_i b_i$$

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compiled with a good compiler, then manually decompiled to C. . .

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

```
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
```

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

Addressing miscompilation

Best industrial practices: more testing; manual reviews of generated assembly code; turn optimizations off; . . .

A more radical solution: 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.*

As a corollary, we obtain:

> *Any safety property of the observable behavior of the source program carries over to the generated executable code.*

An old idea. . .

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

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

An old idea...

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

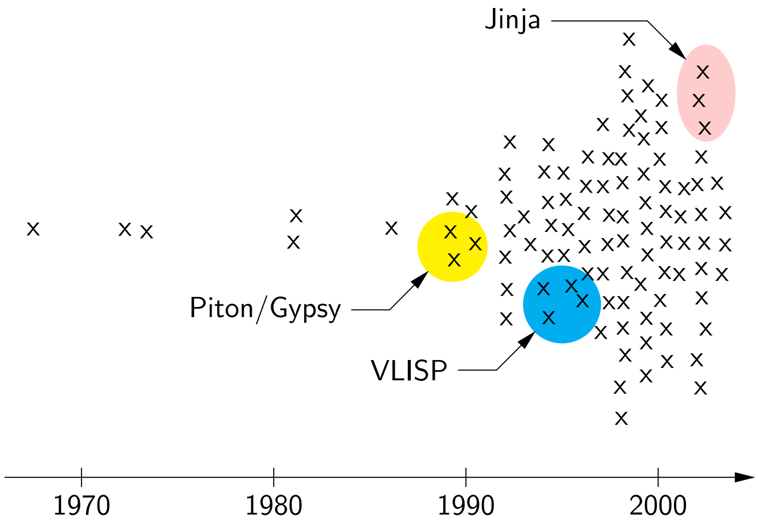
Abstract

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.

The next 100 papers

Maulik Dave, *Compiler verification, a bibliography*, 2003



The CompCert project

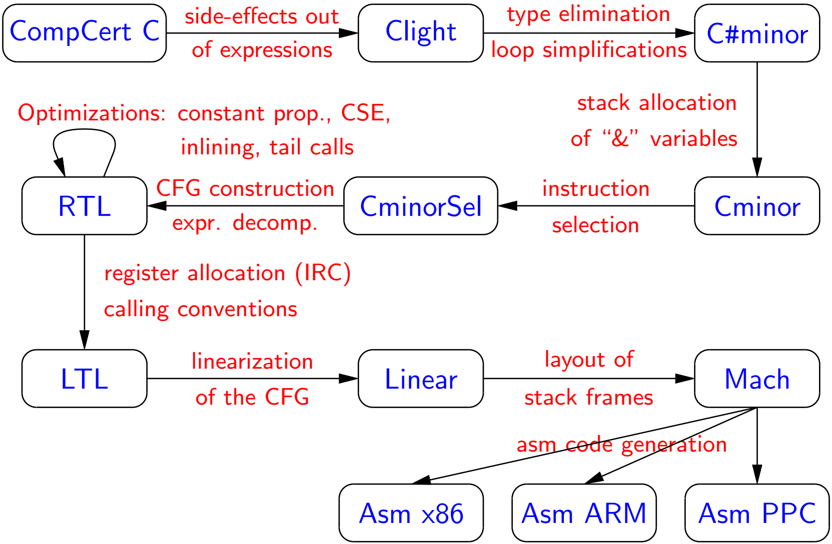
(X.Leroy, S.Blazy, et al)

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

- Source language: a very large subset of C99.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ 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 using Coq

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

```
Theorem transf_c_program_preservation:
  forall p tp beh,
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) beh ->
  exists beh', program_behaves (Csem.semantics p) beh'
            /\ behavior_improves beh' beh.
```

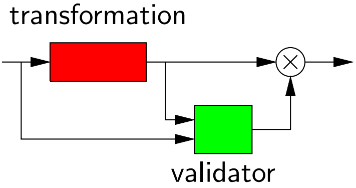
Shows refinement of observable behaviors `beh`:

- Reduction of internal nondeterminism
  (e.g. choose one evaluation order among the several allowed by C)
- Replacement of run-time errors by more defined behaviors
  (e.g. optimize away a division by zero)

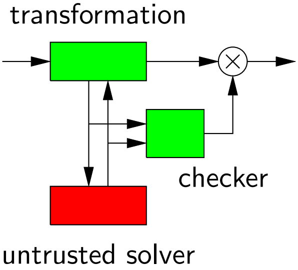
Compiler verification patterns (for each pass)

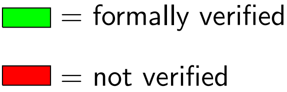
Verified transformation

Verified translation validation



External solver with verified validation





Programmed (mostly) in Coq

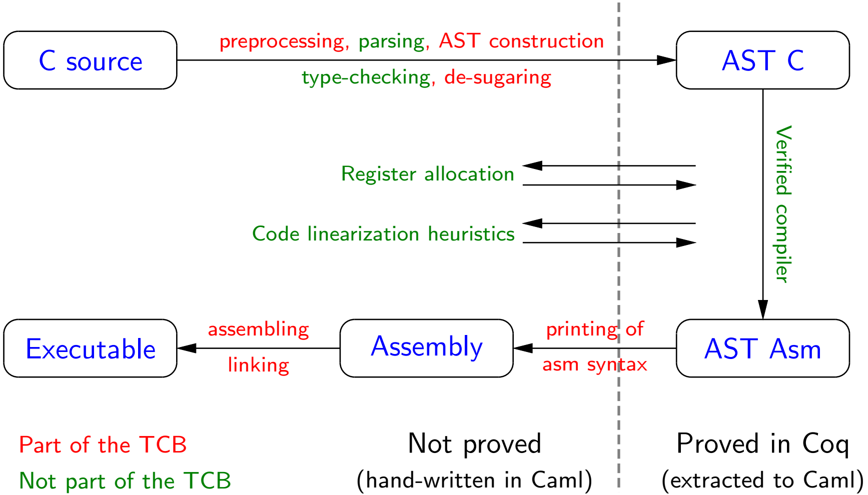
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

- Monads to handle errors and mutable state.
- Purely functional data structures.

Coq's extraction mechanism produces executable Caml code from these specifications.

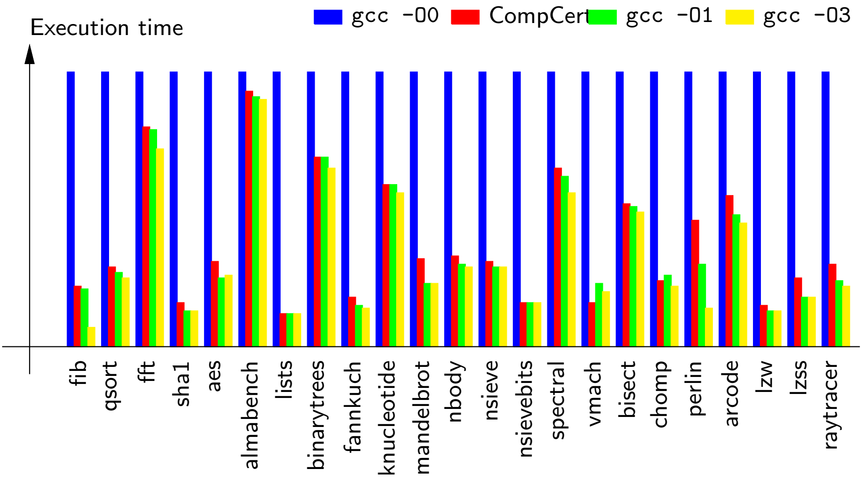
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler



Performance of generated code

(On a Power 7 processor)



A tangible increase in quality

The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011

Part V

Conclusions

Is software perfection within reach?

Perhaps! But at a minimum we need:

- Mathematical specifications (e.g. control-command)
- Appropriate programming languages (e.g. Scade)
- Serious testing (of the airplane kind)
- Formal verification (static analysis, model checking, program proof)
- Trustworthy tools (CompCert, Verasco)
- Theorem proving (Coq, HOL, Z3, . . .)
- . . . and further research!