Formal methods: abstract interpretation & verified compilation

David Monniaux

▶ To cite this version:

David Monniaux. Formal methods: abstract interpretation & verified compilation. Engineering school. Le Kremlin-Bicêtre, France. 2022. hal-04827997

HAL Id: hal-04827997
https://hal.science/hal-04827997v1

Submitted on 9 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal methods: abstract interpretation & verified compilation

David Monniaux

VERIMAG — CNRS

2022-09-06

In a nutshell

Use methods based on mathematics to produce **safe software**.

Contents

Astrée and other analyzers
- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert
- Compiler safety
- Scheduling

Program safety proofs

Prove that a **program** does not end in the **wrong place.**

```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)


***  SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c
```

My industrial involvement

`https://www.astree.ens.fr/`

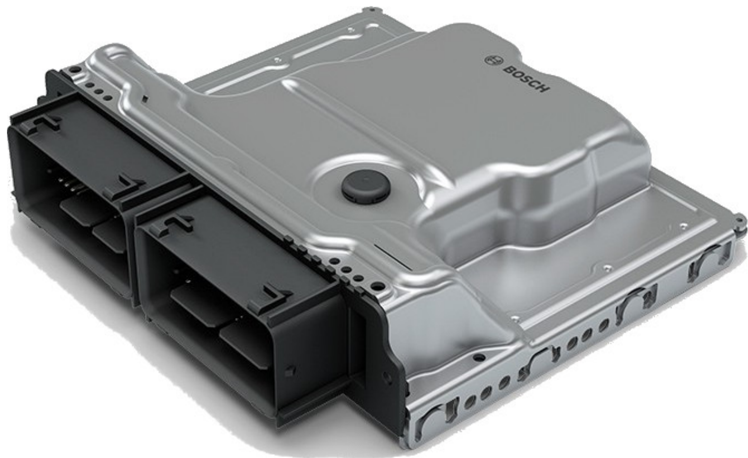
`https://www.absint.com/astree/index.htm`

Use case: safety-critical systems

e.g., fly-by-wire aircraft controls

Then not so critical systems

e.g. car engine control unit



Distinct from testing

Testing

Check a finite number of cases.
What if the bug occurs in other cases (e.g. compiler generates incorrect code for some rarely used block only if some register is X15).

Proof

Prove for all cases.

Safety properties

In this talk: properties of the form
"$\forall$ execution, $\forall$ state along execution, property P holds"

Otherwise said: "the system cannot reach $\neg P$"

Obvious use: $\neg P$ is the **error states**

- runtime errors (division by zero, null pointer, array access out of bounds, invalid cast...)
- assertion violations

First idea: direct proof by induction

1. true when starting the loop
2. if true at iteration n, then true at iteration $n + 1$

Weakness?

A loopy example

```
for(int i=0; i!=100; i++) { }
```

Loop body:

- initialization $i := 0$
- step $i \to i + 1$ if $i \neq 100$

Prove $i < 200$. Inductive?

A loopy example

```
for(int i=0; i!=100; i++) { }
```

Loop body:

- initialization $i := 0$
- step $i \to i + 1$ if $i \neq 100$

Prove $i < 200$. Inductive?

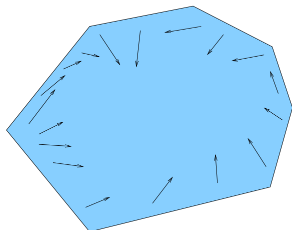
Not inductive. Yet **stronger** $i \leq 100$ inductive!

Strenghtening: executive summary

The property to prove is almost never inductive.

Replace it by a **stronger, inductive property**.

Either ask the user to provide the property (Frama-C, etc.)…
…or have the tool find it (Astrée, etc.)



Contents

Astrée and other analyzers

- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert

- Compiler safety
- Scheduling

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //
  y = x; //
} else { //
  y = -x; //
} //
z = 2*y + x; //
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //
} else { //x ∈ [−7, −1]
  y = -x; //
} //
z = 2*y + x; //
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //
z = 2*y + x; //
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

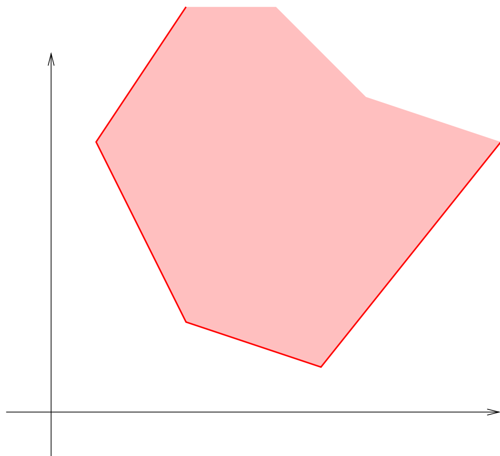
```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Unbounded convex polyhedra



Contents

Astrée and other analyzers
- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert
- Compiler safety
- Scheduling

What must be done

Being able to represent sets of variable / memory states usable in inductive proofs.

Not limited to integers values!

Floating-point

Most abstract domains: ideal mathematics ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$)
Intervals: handle floating-point by directed rounding

Other cases: bound roundoff errors

$x \oplus y = x + y + \epsilon, |\epsilon| \leq \epsilon_r |x + y|$

$x \otimes y = x \times y + \epsilon, |\epsilon| \leq \min(\epsilon_a, \epsilon_r |x + y|)$

ϵ_r "error at last bit of precision"
ϵ_a least positive floating-point value

Data structures

- Point-to graph (may/must)
- Abstract into a single variable
  - all data allocated at single location? (but beware of malloc-like functions)
  - all fields with same field identifier (e.g. in Java)
- Recursive decomposition of memory
- Separation logic?

Contents

Astrée and other analyzers
- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert
- Compiler safety
- Scheduling

Other approaches

Astrée based on abstract interpretation

Other tools based on predicate abstraction, Craig interpolants, etc.

Yet other tools (not discussed here) are **style checkers**.

Polyspace

(commercial) now Mathworks

Astrée

Cousot et al. (commercial)
`http://www.astree.ens.fr/`
`http://www.absint.com/astree/index.htm`

- home-made front-end
- only abstract interpretation
- intervals and octagons
- specialized abstractions for numerical filters
- limited memory abstractions
- now support for parallel programs and C++

CPAChecker

`http://cpachecker.sosy-lab.org/`

- Eclipse CDT front-end
- mostly predicate interpretation
- intervals
- limited support for octagons and polyhedra

PAGAI

`http://pagai.forge.imag.fr/` (Julien Henry)

- LLVM front-end
- uses APRON abstract domains
- path-focusing for SMT-solving
- extra applications to worst case execution time (WCET) analysis

Contents

Astrée and other analyzers
- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert
- Compiler safety
- Scheduling

When proving is too hard

Use technology developed for proving to conduct more extensive testing.
Concolic execution.

e.g. SAGE project at Microsoft `https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf`
automated search for vulnerabilities in MS Office etc.

Contents

Astrée and other analyzers
- Safety properties and induction
- Numerical abstractions
- Other data
- Tools
- Too hard

CompCert
- Compiler safety
- Scheduling

Usual compiler

source code

target code (assembler source, object code, bytecode...)

tens of thousands of bugs in gcc's bug tracker

Why compiler bugs are nasty

A compiler bug may disappear

- if optimization levels are changed to ease debugging
- if a different compiler is used
- if debugging code is added to the program (even just `printf`)

A compiler bug is most often at first **undistinguishable from reliance on undefined behavior** in the program.

CompCert

Formally verified C compiler
project led by Xavier Leroy, then at INRIA, now at Collège de France

Non-commercial `https://github.com/AbsInt/CompCert`
Commercial `https://www.absint.com/compcert/index.htm`

trace of execution = sequence of external calls, volatile read/writes

valid trace of execution at C level

same trace of execution at assembly level

Trace of execution

A compiler optimizer may reorganize everything internally…
but must preserve all **interactions** with the outside world and their **ordering**

- calls to external functions (system calls, I/O, GUI…)
- read/write to volatile variables (for system-level programming)

Use case: traceability

Safe-critical systems (e.g. avionics)

Obligation to match object code to source

Conventional method: `-O0` and some manual inspection

CompCert replaces this by mathematical proofs.
Can **use optimization**.

Semantics and proofs in CompCert

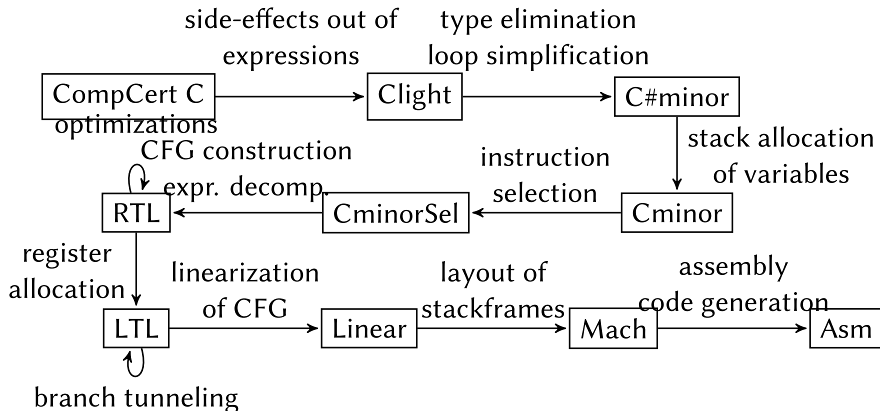
Each intermediate language comes with a **semantics** written in Coq. Gives a **mathematical meaning** to all constructs in the intermediate language.

e.g. at C level, gives a notion of environment of variables (local and global), defines + as addition on various data types depending on types of inputs

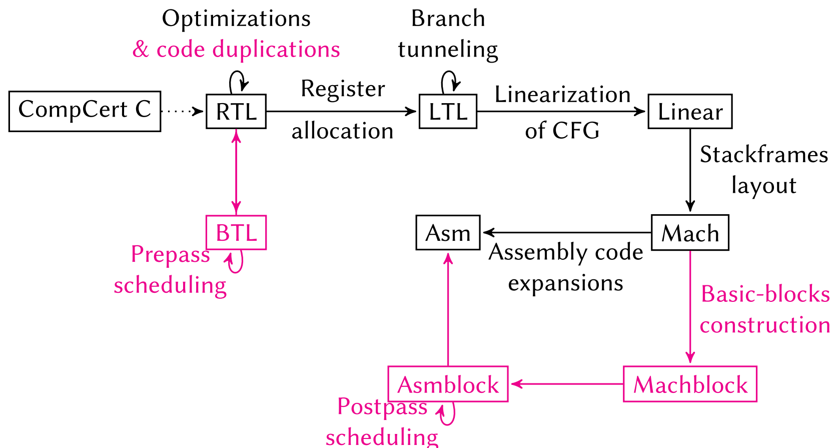
Optimization / transformation phases written in Coq. (Can call external untrusted OCaml code.)

Must prove **simulation** for each phase

Intermediate languages (1)



Intermediate languages (2)



Simulation proofs

Lockstep

one step of program before the transformation

one matching step of program after the transformation

More complex simulations replace sequences of steps by sequences of steps (one step of function call $\rightarrow$ many steps for pushing parameters and calling the function)

Tastes like chicken

Pen-and-paper mathematical proofs can be buggy.

All proofs here checked by **Coq** against the actual compiler code.

`https://coq.inria.fr/`

Overall use

These proofs concern the compiler designer only (and those who file for qualification for safety-critical systems).

For most users, using CompCert is just like using gcc or clang.

```
$ ccomp hello_world.c -o hello_world
$ ./hello_world
```

Or to produce assembly code

```
$ ccomp -S hello_world.c
```

Alternate front-ends

Many high-level languages / domain-specific languages compile to C.

- bugs in the high-level language compiler
- "semantic mismatches" between the high-level compiler and the C compiler

Solution: compile directly to one of CompCert's early intermediate representations!
e.g. **Vélus** `https://velus.inria.fr/` compiles a subset of the synchronous data-flow Lustre / SCADE language used in avionics etc.
and...our current development of a **Rust** front-end

Contents

Astrée and other analyzers
Safety properties and induction
Numerical abstractions
Other data
Tools
Too hard

CompCert
Compiler safety
Scheduling

Scheduling

An example of an **optimization**, particularly for low-power / embedded cores.

Instructions produce outputs a number of clock cycles after receiving their inputs.
Take this into account to **schedule** instructions.

A menu

1. oysters
2. veal blanquette
    2.1 prepare blanquette
    2.2 cook it
3. millefeuille
    3.1 **puff pastry**
        3.1.1 fold 1, wait 30 minutes
        3.1.2 fold 2, wait 30 minutes
        3.1.3 fold 3, wait 30 minutes
        3.1.4 fold 4, wait 30 minutes
        3.1.5 fold 5
    3.2 cream

Scheduling

"Official" CompCert produces instructions roughly in the source ordering.

Not the best execution order in general!

Especially on in-order cores.

Our solution: **verified scheduling**

Superblock scheduling

1. Partition each function into **superblocks**: one entry point, possibly several exit points, no cycle
2. Possibly do some other reorganization: tail duplication, etc. to get bigger superblocks
3. Schedule the superblock (no proof needed)
4. Witness through symbolic execution that the original and scheduled superblocks have equivalent **semantics** (proof needed)

Before register allocation, on IR.

On Kalray KVX and AArch64: reschedule basic blocks on assembly instructions after register allocation.

Equivalent semantics

- Same order of exit branches in original and scheduled superblock
- All live pseudo registers and memory have the same value at same exit point (non-live registers can differ)
- Same (or smaller) list of instructions that may fail (division by zero, memory access) reached at same exit point

Obtained by **symbolic execution**: two registers are considered equal if computed by exactly the same symbolic terms

Example

$r_1 := a * b$
$r_3 := a - b$
$r_2 := r_1 + c$
branch($a > 0$, EXIT1)

$r_3 := a - b$
$r_4 := a * b$
$r_2 := r_4 + c$
branch($a > 0$, EXIT1)

r_1 and r_4 are both dead at EXIT1 and at final point.

These two blocks are **equivalent**: in both cases
$r_2 = (a * b) + c$ and $r_3 = a - b$

Information needed

For all instructions

- **latency**: clock cycles between consuming operands and producing the value (or, more generally, a timetable of when each operand is consumed after the instruction is issued)
- **resource consumption**: CPU units in use that preclude other instructions being scheduled at the same time

Very difficult to find even for "open cores"!!!
(Reverse-engineer gcc and LLVM?)

Online

https://www.absint.com/compcert/index.htm
https://github.com/AbsInt/CompCert

Our version of CompCert with optimizations not found in the "official" releases + the KVX target:
https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx

Pre-pass scheduling: KVX; Cortex-A53/A35 (AArch64); Rocket, SweRV EH1 (Risc-V)
Post-pass scheduling: KVX; Cortex-A53/A35 (AArch64)

Verimag

https://www-verimag.imag.fr
https://www-verimag.imag.fr/~monniaux/