Software Verification for Fun and Profit

David Monniaux

VERIMAG

May 6, 2015

Outline

This talk in 20 seconds

Executive summary:

- bugs are a real problem
- proving their absence is hard
- one should not despair because of **undecidability** or **NP-completeness**
- sometimes **simple solutions** work well!

Outline

Ariane 5

Maiden flight (501) of Ariane 5 (1996)

Ariane 5

Maiden flight (501) of Ariane 5 (1996)

Ariane 5, explanation

Reason:

- some software designed for Ariane 4 was reused in Ariane 5, a larger rocket: physical value ranges were different
- a conversion from 64-bit floating-point into 16-bit signed integer value **overflowed**
- this conversion was not protected, resulting in an exception
- it was in a part of the software not even needed for Ariane 5 at this point of the flight sequence!
- the SRI computer shut down
- **the rocket had to be destroyed**

Why Ariane's engineering failed

Redundancy: there were two identical computer systems... but

Why Ariane's engineering failed

Redundancy: there were two identical computer systems... but

"The reason behind this drastic action lies in the culture within the Ariane programme of only addressing **random hardware failures**. From this point of view exception — or error — handling mechanisms are designed for a random hardware failure which can quite rationally be handled by a backup system."

Two identical systems with buggy software may both fail for the same reason!

Boeing 787

May 1st, 2015, US Federal Aviation Authority airworthiness directive:

"This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a ***software counter*** *internal to the GCUs that will* ***overflow after 248 days of continuous power****. We are issuing this AD to prevent* ***loss of all AC electrical power, which could result in loss of control of the airplane****."*

Boeing 787

May 1st, 2015, US Federal Aviation Authority airworthiness directive:

"This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a ***software counter*** *internal to the GCUs that will* ***overflow after 248 days of continuous power****. We are issuing this AD to prevent* ***loss of all AC electrical power, which could result in loss of control of the airplane****."*

Note: 248 days $\simeq 2^{31} \times 10^{-2}$ s, suppose 100 Hz clock overflowing

Boeing 787, solution

Boeing 787, solution

"Repeat the electrical power deactivation thereafter at intervals not to exceed **120 days**."

Also known as "reboot the machine often enough".

Heartbleed

2014 bug in OpenSSL in the implementation in Heartbeat extension to SSL (“secure connections”)

No proper array bound checks on a buffer
→ **attacker can read** chunks of memory **secret data**

Other examples

Therac-25 (1985–1987) A faulty radiation therapy series of machines **kills 3** and harms 3 at least.

Cause:

Other examples

Therac-25 (1985–1987) A faulty radiation therapy series of machines **kills 3** and harms 3 at least.

Cause: **race condition**

Patriot missile (1991) A Patriot anti-missile missile fails to destroy an Iraqi Scud missile. **28 soldiers die**.

Cause:

Other examples

Therac-25 (1985–1987) A faulty radiation therapy series of machines **kills 3** and harms 3 at least.

Cause: **race condition**

Patriot missile (1991) A Patriot anti-missile missile fails to destroy an Iraqi Scud missile. **28 soldiers die**.

Cause: **clock drift** after being turned on for an unusual length of time (solution: reboot the computers)

Bugs do occur

Stupid, basic bugs still occur!

...but they are hidden in massive amounts of code!

How about more clever bugs? E.g. wrong algorithms?
(some algorithms are sometimes proved incorrect years after publication)

High-tech, high-assurance solution

Can we **search** for bugs?

Can we **prove** the absence of bugs?

I'll cover both

- search for bugs in finite depth
  (focus: worst-case execution time)
- proof of absence of bugs: invariant inference

Outline

Traditional safety-critical systems

Airplanes Fly-by-wire controls, inertial guidance, FADEC

Trains Signaling systems, automated driving

Cars Fuel injection & ignition, brake-by-wire, steering-by-wire

Infusion pumps Software control

US FDA improvement initiative

"Infusion pumps have been associated with persistent safety problems that can result in over- or under-infusion, and missed or delayed therapy."

Security problems

- Previously, browser or server bugs could result in loss of personal information, loss of credit card numbers (money)
- How about a Heartbleed-like bug in a browser or application used by a dissident in an authoritarian country?

Outline

What Is to Be Done?

Better languages? For some “stupid” bugs (buffer overflows, arithmetic overflows...), perhaps.

Coding practices Coding standards, code reviews, etc.

Testing Did not catch Heartbleed etc. On airplanes etc., mostly successful but very costly.

Program proofs and “formal methods”

Formal methods?

1. Attach a mathematical meaning to the program ("semantics")
   e.g. "+ here means + over 32-bit unsigned integers"
   (very tricky for real-life languages, e.g. C)
2. Prove properties over it.
3. (optional) Validate the proof with a small trusted computing base

Successes of formal methods

A very partial list of semi-automated tools:

B Method	"Meteor" project for line 14 of the Paris metro
HOL Light	Proofs on Intel hardware (floating point) Flyspeck project (proof of Kepler's conjecture)
ACL2	Proofs on AMD hardware
Isabelle	Flyspeck project (proof of Kepler's conjecture)
Coq	Proof of the four-colour theorem (graph colouring) Proof of the Feit-Thompson theorem (every finite group of odd order is solvable) **CompCert**, certified C compiler (proof that it either fails or compiles C into correct object code)

Successes of formal methods

A very partial list of automated tools:

Polyspace Start-up formed after the Ariane explosion. Located near INRIA-Montbonnot. Later bought by The Mathworks.
Automated proofs of absence of runtime errors for embedded Ada, C, C++ programs.

Astrée Automated proofs of absence of runtime errors for C programs.
Developed at CNRS / ENS-Paris.
Marketed by Absint GmbH.
Used by e.g. Airbus A340, A380 and following

Frama-C Semi-automated and fully automated proofs of absence of runtime errors and respect of specifications
Developed at CEA LIST, INRIA Saclay, LRI (CNRS / Université Paris Sud)

Recent success: bug in Python & Java standard libraries

Timsort (2002 sorting algorithm) implemented in Python, OpenJDK, Android contains a bug.

Bug found when attempting to prove its correctness in KeY.

Outline

Undecidability

For any nontrivial class of programs, deciding **halting** or **any final property of the execution** is undecidable.

Rice's theorem

In layman's terms: **no algorithm** for deciding properties

1. that are nontrivial (trivial = "all programs accepted", "no program accepted")
2. on the **final result of programs** (unbounded execution time)
3. over programs with **unbounded memory**
4. with no **false positives**
5. with no **false negatives**
6. and **always terminating**.

These conditions leave research directions open!

But memory is finite!

Turing and Rice's results apply to unbounded memory.
Physical systems have bounded memory.

Implicit-state or explicit-state **model-checking** deal with finite state systems.

But memory is finite!

Turing and Rice's results apply to unbounded memory.
Physical systems have bounded memory.

Implicit-state or explicit-state **model-checking** deal with finite state systems.

If program represented as "transition relation" over a vector of bits, the **reachability** problem is **PSPACE-complete**.

PSPACE-complete conjectured to be harder than NP-complete.

All "practical" algorithms for reachability use $\Theta(2^n)$ time and memory in the worst case.

$n \simeq 2^{35}$ on this machine.

Almost all software systems should be treated as infinite-state for practical purposes.

Outline

But all programs have loops!

- **Search** for bugs at bounded depth (bounded model checking).
- As building block for other analyses.

Outline

A simple model

- Scalar ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, bitvector) variables
- Linear arithmetic
- If-then-else

```
int x, y, z;
x = any_int();
y = any_int();
assume(x >= -5 && x <= 5);
assume(y >= -10 && y <= 10);
if (x <= y) {
  z = x-y;
} else {
  z = x+y;
}
assert(z >= -15 && z <= 15);
```

Translation to satisfiability

assertion violated $\iff$ formula satisfiable

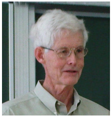
$$
\begin{aligned}
&(x \geq -5 \wedge x \leq 5) \wedge (y \geq -10 \wedge y \leq 10) \wedge \\
&\qquad\qquad \big((x \leq y \wedge z = x - y) \vee (\neg(x \leq y) \wedge z = x + y)\big) \wedge \\
&\qquad\qquad\qquad\qquad \neg(z \geq -15 \wedge z \leq 15)
\end{aligned}
$$

incorrect execution $\equiv$ satisfying assignment

Satisfiability testing

A quantifier-free formula with $\wedge$, $\vee$, $\neg$

Booleans only Classical SAT problem **NP-complete**

Booleans + linear arithmetic on $\mathbb{R}$ or $\mathbb{Q}$ NP-complete

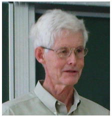
(Booleans +) Linear arithmetic on $\mathbb{Z}$ NP-complete (pure satisfiability problem in integer linear programming)

(Booleans +) Polynomial arithmetic on $\mathbb{R}$ NP-hard, exponential algorithms

Satisfiability testing

A quantifier-free formula with $\land$, $\lor$, $\neg$

Booleans only Classical SAT problem **NP-complete**

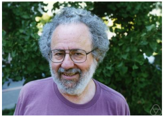
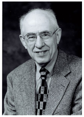
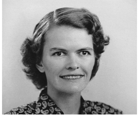
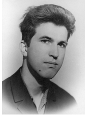
 

Booleans + linear arithmetic on $\mathbb{R}$ or $\mathbb{Q}$ NP-complete

(Booleans +) Linear arithmetic on $\mathbb{Z}$ NP-complete (pure satisfiability problem in integer linear programming)

(Booleans +) Polynomial arithmetic on $\mathbb{R}$ NP-hard, exponential algorithms

Polynomial arithmetic on $\mathbb{Z}$ undecidable (Hilbert's Tenth Problem)

In popular usage

"Satisfiability modulo theory" (SMT) solvers are tools that

1. take as input a formula (often quantifier-free) over a theory (e.g. linear real arithmetic)
2. give a model if satisfiable
3. answer "unsatisfiable" otherwise

Examples include

- Microsoft Z3 (now under MIT Free License)
- Yices
- MathSat
- CVC4
- Boolector
- Alt-Ergo

How SMT-solvers work

$$(x \geq -5 \wedge x \leq 5) \wedge (y \geq -10 \wedge y \leq 10) \wedge \\ \big((x \leq y \wedge z = x - y) \vee (\neg(x \leq y) \wedge z = x + y)\big) \wedge \\ \neg(z \geq -15 \wedge z \leq 15)$$

Backtracking search by assigning truth values to **atomic propositions** syntactically present in formula.

Learning:

- pure SAT "constraint-driven clause learning" (CDCL)
- **theory lemmas** e.g. $\neg(x \leq y \wedge y \leq z \wedge z < x)$

SMT-solvers in use

In proof assistants e.g. Isabelle “sledgehammer”

In semi-automated program provers e.g. Frama-C

In automated program analysis e.g. Pagai, UFO, CPAChecker

Bounded model checking CBMC

In fuzzing e.g. Microsoft SAGE

Limitations

Computability Some classes of formulas are undecidable (e.g. with quantifiers and uninterpreted functions)

Complexity **NP-hardness** or worse.

Hope that the backtracking strategy blocks out the search space fast enough!

Sometimes...it just blows up!

Outline

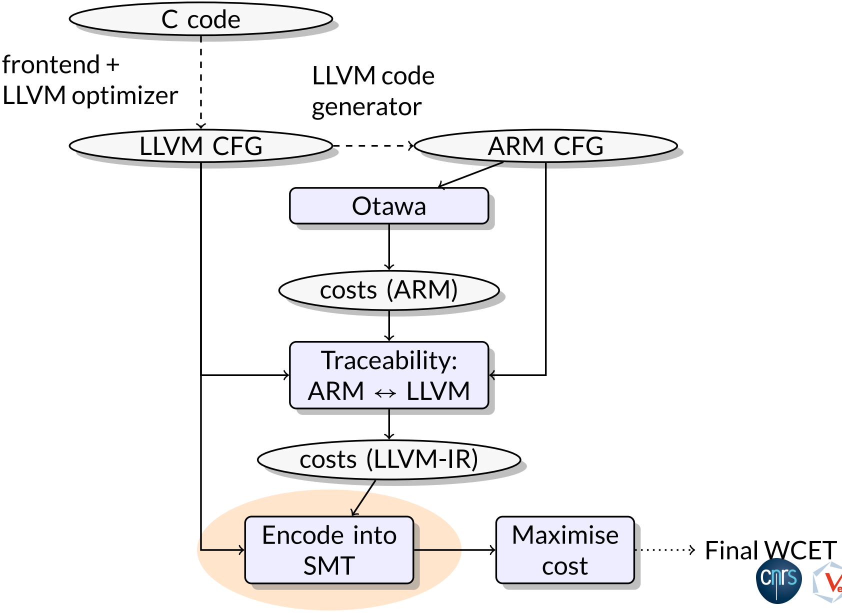
Worst-case execution time by SMT

Encode loop-free program into formula as before

Solutions are execution traces, special variable *cost*

Minimize *bound* by successive queries “is there a trace with $\mathit{cost} \geq \mathit{bound}$”? (binary search)

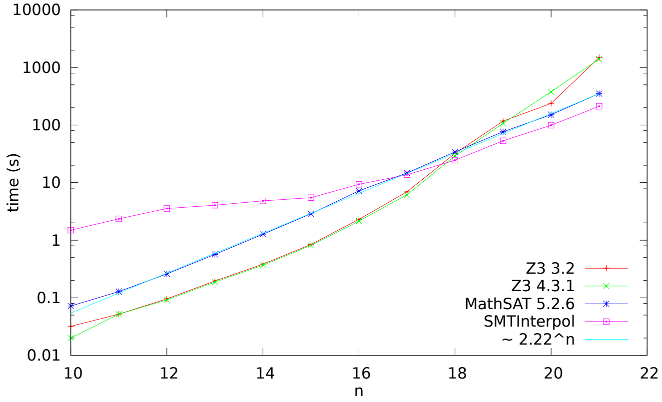
Our Workflow



Proving optimality is costly

Last test in binary search:
proving that there is no trace longer than *bound*

Very simple examples, sequence of $2n$ if-then-else's.



The simple example

```
bool b₁ = any_bool(), ..., bₙ = any_bool();
if (b₁) { /* timing = 2 */ }
      else { /* timing = 3*/ }
if (b₁) { /* timing = 3 */ }
      else { /* timing = 2*/ }
...
if (bₙ) { /* timing = 2 */ }
      else { /* timing = 3*/ }
if (bₙ) { /* timing = 3 */ }
      else { /* timing = 2*/ }
```

Explanation

All current production-grade SMT solvers use DPLL(T) scheme: search

- over the atomic propositions syntactically present
- with backtrack only when a conjunction of arithmetic propositions is unsatisfiable

On this example, this leads to **exponential proofs**.
And thus **exponential time**.

Moral and solution

Exponential time does occur on relevant examples, not just academic concocted examples.

What Is To Be Done?

Moral and solution

Exponential time does occur on relevant examples, not just academic concocted examples.

What Is To Be Done?

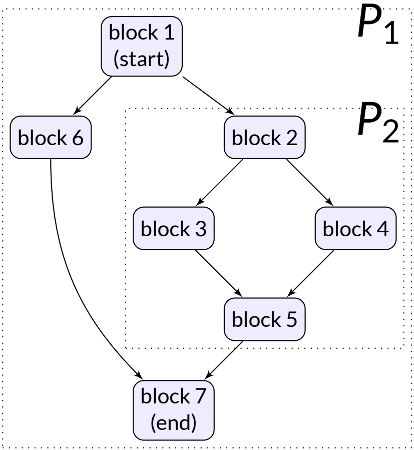
Introduce "cuts" $C_1, \dots, C_n$ that enrich the set of atomic propositions present:

Replace formula F by $F \wedge C_1 \wedge \cdots \wedge C_n$ where $F \Rightarrow C_1 \wedge \cdots \wedge C_n$

In our case

The "cuts" or "summaries" C_i are

"The total time spent in this part of the program is always $\leq$ XXX"



Experiments with ARMv7

OTAWA for Basic Block timings
PAGAI for SMT, see `pagai.forge.imag.fr`, uses Z3 SMT solver

Benchmark name	WCET bounds (#cycles) Otawa	SMT	diff	Analysis time (s) with cuts	no cuts	#cuts
statemate	3297	3211	2.6%	943.5	$+\infty$	143
nsichneu (1 iteration)	17242	13298	22.7%	6hours	$+\infty$	378
cruise-control	881	873	0.9%	0.1	0.2	13
digital-stopwatch	1012	954	5.7%	0.6	2104.2	53
autopilot	12663	5734	54.7%	1808.8	$+\infty$	498
fly-by-wire	6361	5848	8.0%	10.8	$+\infty$	163
miniflight	17980	14752	18.0%	40.9	$+\infty$	251
tdf	5789	5727	1.0%	13.0	$+\infty$	254

Moral and future work

We know the **structure** of our optimisation problem.

A naive encoding into a NP-complete satisfiability problem leads to exponential solving.

A redundant encoding helps the solver avoid exponential behaviour.

Current work: detect this directly in the solver, **without using the structure** of the original problem.

(Automatic cut generation, extended resolution)

Summary

- NP-complete
- if exponential behaviour detected, **understand** why
- find a class of nasty formulas
- derive a workaround (clever encoding, detection inside the solver)

Outline

Programs with loops

How do we deal with loops?

(and “goto”, and recursion, etc.)

Floyd-Hoare proofs

Require the user to supply **inductive invariants**:

- which hold initially
- assume it holds at the beginning of iteration n, still hold at iteration $n + 1$

Hold **by induction** at every iteration.

Frama-C

```
/*@ requires
  @   n >= 0 && \valid(t+(0..n-1)) &&
  @   \forall int k1, k2; 0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @ assigns \nothing;
  @ ensures
  @   (0 <= \result < n && t[\result] == v) ||
  @   (\result == -1 && \forall int k; 0 <= k < n ==> t[k] != v);
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  /*@ loop invariant
    @   0 <= l && u <= n-1
    @   && (\forall int k; 0 <= k < n ==> t[k] == v ==> l <= k <= u) ;
    @ loop assigns l,u ;
    @ loop variant u-l ;
    @*/
  while (l <= u ) {
    int m = l + (u-l) / 2;
    //@ assert l <= m <= u;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

Floyd-Hoare logic

Checking inductiveness ≡ checking

```
assume(invariant);
assume(loopcondition);
LOOP BODY
assert(invariant);
```

Reduces to loop-free programs!

Difficulties

Annotating programs with invariants is cumbersome.

How to **automatically infer the invariants**?

Strong vs weak invariant

weakest
: "anything is possible" (useless)

strongest
: exact description of reachable states: complicated (and in undecidable class)

Two kinds of approaches:

- guided by a property to prove
- unguided: try to find a "strong" invariant

CEGAR: Guided by a property

"**C**ounter**E**xample **G**uided **A**bstraction **R**efinement" (CEGAR)

From proofs that error states are unreachable by longer and longer counterexample traces…

generalize to an inductive argument!

(Complicated: uses Craig interpolants, refinements, etc.)

I do not do that!

Abstract interpretation

Idea: look for inductive invariants in a restricted class of properties ("abstract domain")

Numerics

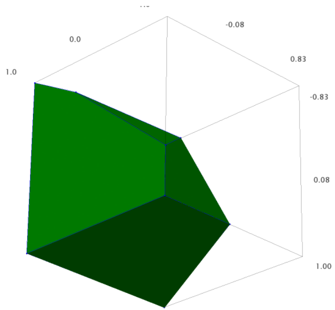
- products of intervals e.g. $(x, y) \in [0, 2] \times [3, 5]$
- difference bound matrices e.g. intervals + constraints $x - y \leq 6$
- convex polyhedra , e.g. $2x + 3y \leq 4 \wedge x \geq 3 \wedge y \geq 5$

Data structures

- tree automata, forests of trees
- alias sets
- array abstractions

and much more!

Polyhedra



Representations with vertices and/or constraints

At VERIMAG: project VERASCO, library VPL

How about keeping it simple?

Polyhedra are expensive.

How about mere intervals?

Example with intervals

```
int[] t = new int[140];
for(int i=0; i<t.length; i++) t[i]=42;
```

Example with intervals

```
int[] t = new int[140];
for(int i=0; i<t.length; i++) t[i]=42;
```

With explicit bound checks:

```
int[] t = new int[140];
for(int i=0; i<t.length; i++) {
  if (i<0) throw new
     ArrayIndexOutOfBoundsException();
  if (i>=t.length) throw new
     ArrayIndexOutOfBoundsException();
  t[i]=42;
}
```

Problem of inductiveness

```
int[] t = new int[140];
for(int i=0; i<t.length; i++) t[i]=42;
```

Find inductive invariant $i \in [l, h]$

Initiation $l \leq 0 \leq h$

Inductive step $h \geq \min(139, h) + 1$

Solving inductiveness

Approximate solving with "widening"

- Run interval propagation: $[0, 0], [0, 1], [0, 2]$
- Extrapolate to $[0, \infty)$
- Check for inductiveness
- Refine to $[0, 140]$

Exact solving Several approaches

- acceleration
- policy iteration

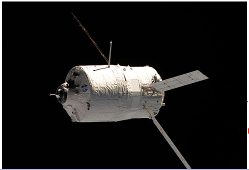
Astrée analyzer

http://www.astree.ens.fr/
http://www.absint.de/astree

- Keep it simple: interval analysis, approximate solving with widening and refinement
- On top of it: trace partitioning (distinguish paths, sometimes) etc.
- And **special abstractions** for filters used in control applications

Can prove the absence of runtime errors in large safety-critical programs, e.g. fly-by-wire controls





Invariant inference: executive summary

Scalability varies from high (e.g. interval analysis) to low (e.g. certain acceleration approaches, some predicate abstractions...)

Precision varies across applications:
lower ratio of "false positives", or true properties that the tool fails to prove.

Scalability and precision greatly improved by tuning for a **class** of applications and properties, e.g

- API compliance in device drivers (e.g. Microsoft Device Driver Verifier)
- absence of runtime errors in embedded control applications

Outline

Key insights (scientific)

One should not blindly fear

- undecidable problems
  $\rightarrow$ often simple arguments work
- NP-hard problems
  $\rightarrow$ efficient pruning of the search space

Sometimes linear complexity is just too much (e.g. each operation costs $\Theta(n)$ where n is the total number of variables in the program)

$\rightarrow$

Prune down the problems to reduce excessive complexity

Worst-case complexity is not necessarily meaningful

Key insights (industrial use)

Generic tools can be expected to have mediocre performance

- too many alarms (properties that cannot be proved)
- excessive complexity

Much better results if tools adapted to uses

- identify key problems (e.g. bad invariants from certain constructs, exponential behaviours)
- generalise them
- solve the generalisation (e.g. filter analysis, cuts)

But industry, too often

- refuses to give examples
- stops after first attempt with off-the-shelf generic tool

More information on:
`http://www-verimag.imag.fr/~monniaux/`
`http://verasco.imag.fr/`
`http://stator.imag.fr/`

European Research Council
Established by the European Commission

Current research:

- formal (Coq) proofs of analysis tools
- extended resolution and "summaries" in satisfiability testing
- alternative inductive invariant inference
- combinations of abstraction and exact solving
- combinations of numeric and "discrete" case analysis
- analysis of properties array and data structures by source-to-source abstractions