Scaling up with abstract interpretation

Patrick Cousot and Radhia Cousot

Usable Verification
Savannah, GA
Jan. 20, 2009

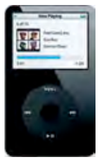
Bugs are also everywhere


"The Australian Transport Safety Bureau (ATSB) found that the main probable cause of this incident was a *latent software error* which allowed the ADIRU to use data from a failed accelerometer"

http://www.atsb.gov.au/newsroom/2008/release/2008_43.aspx,
http://en.wikipedia.org/wiki/Qantas_Flight_72

Safety/mission critical software is everywhere

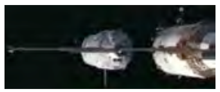










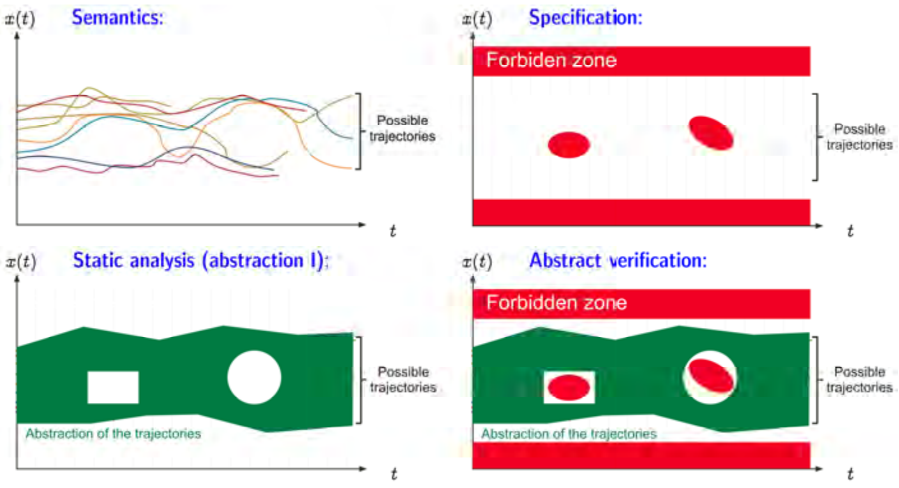


etc...

Verification versus debugging

- Unsound debugging: testing, bug-pattern finding, bounded model-checking,...
  - not difficult, useful, and so popular
  - scales up easily
- Sound verification: deductive methods, exhaustive abstract model-checking of safety properties, static analysis,...
  - useful, difficult, and so rare
  - ultimately indispensable for safety/mission criticality
- [Un]soundness should be clearly stated

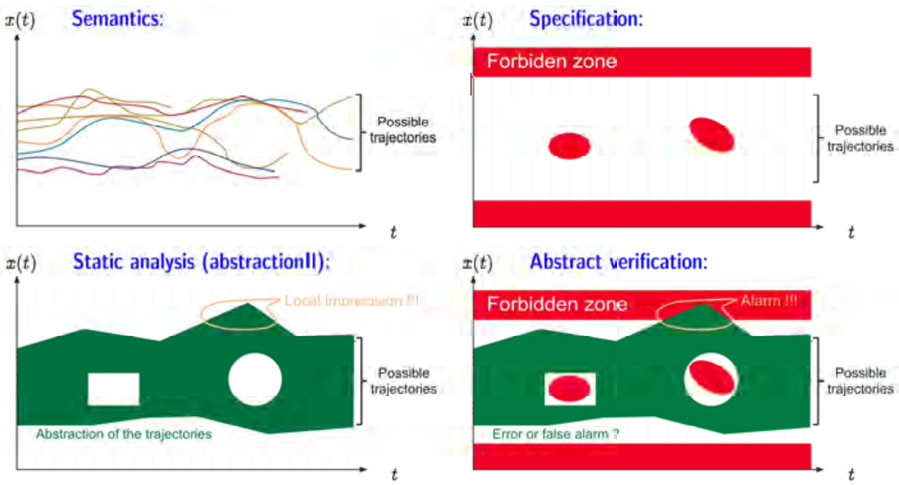
Principle of abstraction-based verification



COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4^{th} *POPL* (Los Angeles, 1977), 238–252.

OK

Principle of abstraction-based verification (cont'd)



COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4^{th} *POPL* (Los Angeles, 1977), 238–252.

False alarm

Abstract interpretation-based static verification is successful in industry

- Polyspace verifier (The MathWorks)
- Stack analyzer (AbsInt)
- AiT (WCET analyzer by AbsInt)
- Astrée:
  - Sound
  - Scales to 10^6 LOCs of C code
  - Effective (5mn to 35h) with no false alarms for control/command applications
  - To be commercialized by AbsInt

The difficulty of program verification

- Relatively "easy" in the small:
  - By exhaustive enumeration (e.g. model-checking)
  - User guided deductive proofs (e.g. proof checkers)
- Very difficult in the large:
  - Safety/mission critical software is routinely millions of lines
  - Approximate abstractions are necessary
  - Bug-finding helpful but unsatisfactory
  - False alarms as a result of undecidability

How to scale up ?

I don’t know (1)

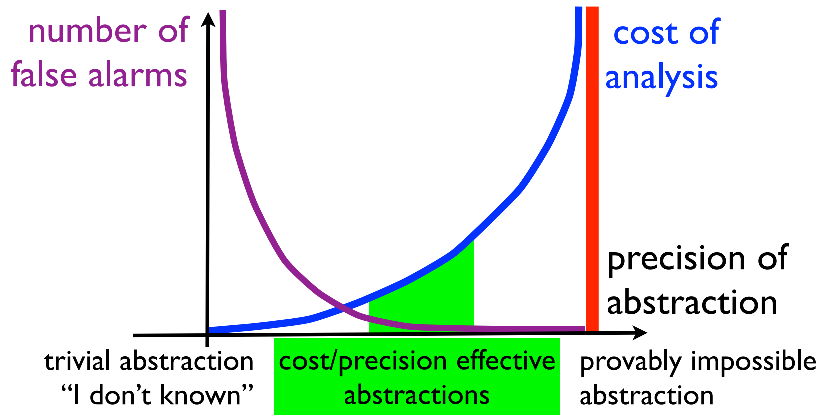
A few modest suggestions only!

(1) typical answer in abstract interpretation

I don’t know (1)

(1) typical answer in abstract interpretation

Choose the right abstraction for the right problem



Consider modularity (of the analysis)

- Analyze programs by parts
  - may be difficult to discover the parts
  - may be difficult to discover the interfaces
  - needs costly relational analyzes
- Analyze program globally
  - simpler abstractions
  - which can be very efficient
- Program parts may unavailable → stubbing & input configuration is necessary

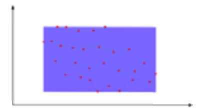
Choose efficient representations of abstract properties

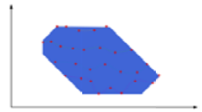
- General representations: formulae in logic/theorem provers, BDDs in model-checking, bit-vectors in dataflow analysis, ...
  - no universal encoding of data does always scale up in algorithmics
- Abstraction specific representations:
  - efficient algorithms require adequate data representations

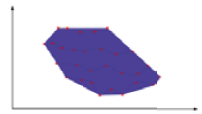
Consider modularity (of the analyzer)

- Can be very hard to guess the right abstraction:
  - experimentation is required to find the appropriate cost/precision balance
  - so easy modifiability of the verifier is indispensable
- Extensible modular verifier:
  - many (parametric) abstractions
  - abstractions are modular with common interfaces
  - abstractions can be inserted/changed/replaced
  - abstractions can be combined

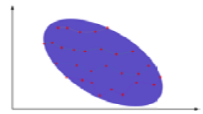
Combine universal & domain-specific abstractions

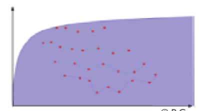






- Universal abstractions:
  - designed once and for all
  - useful as an everywhere usable basis
  - can produce many false alarms (e.g. Polyspace verifier)
  - acceptable when over-approximation is acceptable (compiler optimization, WCET analyzer, etc.)





- Domain/problem-specific abstractions:
  - designed 'on demand'
  - very precise
  - necessary to reach no false alarm

Consider local (better than global) abstractions

- Global abstraction: same abstraction is used everywhere in the program, e.g.
  - Data flow analysis
- Local abstraction: different abstractions are used in different program regions (depending on estimate of required precision) e.g.
  - Manual proof
  - Astrée
- Necessary to balance cost/precision

Refine by adjusting and combining abstractions

- Manuel refinement can be intelligent
- User-guided refinement through directives:
  - Ajust precision of parametric abstractions
  - Hints to analyzer on where to use which abstractions
- Designer refinement:
  - Add new abstractions combined with existing ones to enhance precision
  - The analyzer must be designed to be modular

Understand the sources of imprecisions

- The weaskest inductive argument necessary to make the proof is not expressible within the abstraction
- No way out of a required refinement
- Automatic refinement does not scale
  - Requires to go to the concrete semantics
  - The most abstract refinement is fixpoint computable but not convergent
  - Ultimately equivalent to a fixpoint computation in the concrete

Conclusion

To scale, you must be quasi-linear!

The end, thank you for your attention