Verification of Embedded Software: Problems and Perspectives

Patrick COUSOT
École Normale Supérieure
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
www.di.ens.fr/ cousot

Radhia COUSOT
École Polytechnique
91128 Palaiseau cedex, France
Radhia.Cousot@polytechnique.fr
lix.polytechnique.fr/ rcousot

EMSOFT'01, Lake Tahoe, CA, U.S.A. October 8–10, 2001

Software Quality

- Exponential complexity growth in VLSI with decreasing or constant costs;
- Corresponding proportional growth in software (maybe with a delay of few months or years);
- An operating system running a large number of applications presently crashing every 24 hours, will crash:
  - every 30 minutes within a decade,
  - every 3 minutes if the software size is multiplied by 10.

→ Hardly acceptable for safety critical systems!

Introduction on Formal Methods

This work was supported in part by the RTD project IST-1999-20527 Daedalus of the european IST FP5 programme.

Success Stories for Formal Methods: (1) Theorem Proving Based Deductive Methods

Embedded software for the driverless Meteor line 14 metro in Paris (after failure in Lyon):

- B specification of 115 000 lines;
- compiles into a 87 000 lines ADA program;
- correctness proof, using interactive theorem proving, required to handle manually 27 800 proof obligations;
- 1400 rules had to be added to the prover and proved correct (900 of which automatically);

- No error was ever found in the embedded software nor in its B specification;
- All errors where found at the interfaces not satisfied by the central control software (not developped in B);
- Expansive: 600 person/years!

- State explosion problem: still has to scale up for hardware, not speaking of software:
  - Evolve from debugging to formal verification;
  - Human-understandable temporal specifications;
  - Automatize the design of models.

Success Stories for Formal Methods: (2) Model-Checking

- Most hardware design companies now have model-checkers (after the famous FDIV design fault in the Pentium processor);
- Can verify circuit designs of a few hundreds/thousands of registers (with abstraction of their surrounding environment);

Success Stories for Formal Methods: (3) Program Static Analysis

After the ARIANE 5 flight 501 failure [1]:

- The error was caught (too late!) by an abstract interpretation based static analysis of the program;
- Other errors showed up (data races, divisions by zero, etc.);

---

[1] due to the inertial reference system sending incorrect data following a software exception resulting from an unprotected data conversion from a too large 64-bit floating point to a 16-bit signed integer value

- Static analysis relies on an abstract model of the program semantics;
- The precision of the approximation can be tailored to the available time/memory resources;
- Very precise abstractions are suitable for small programs (few thousands of lines) but global analysis of very large programs (millions of lines) require loose abstractions.

Challenges for Verification Techniques

- Software verification cost is well-known to be non-linear in the software size;
- So informal and formal verification techniques must scale up at a much higher rate that hardware evolution;
- We highlight some of the verification problems to be considered ;
- We envision possible abstract interpretation based program static analysis solutions.

Warning

This presentation is more a wish list on present or future work for opening discussion rather than a technical contribution.

Challenges in Embedded Software Verification

Software Models

Programming Language Semantics Abstraction

- Program analysis is based on abstractions of the programming language semantics;
- Abstract interpretation provides a mathematically safe approximation methodology;
- The model of the program to be verified is provided automatically to the user;

- Many difficulties:
  - The execution environment (including operating system) must be formalized and abstracted;
  - Programming language standards are often very informal;
  - Most standards are continuously revised;
  - Most compilers do not strictly implement standards;
  - Libraries often have no formal specification/semantics;
- Challenge: design stable programming languages semantics which are usable for program verification and enforcable in portable implementations.

Program Specific Abstractions

- There always exists a complete finite approximation to prove a given specification of a given computer system semantics;
- Discovering this abstraction to a finite model is logically equivalent to a formal correctness proof;
- Hand made abstractions are very difficult to design even for small or medium size programs (few hundred thousands lines).

Abstraction in Model Checking

- Three/four different descriptions of the real-world system or program:
  1. in a programming language for the implementation;
  2. in a verification language for the model;

  (3. in an abstract verification language for the finite abstract model;)

  4. in a logic language for the specification of the properties of the model which have to be checked.

- The formal verification is between the model and its specification;

A few neglected difficulties:

- How formal is the relation between the concrete and abstract models?
- How formal is the relation between the concrete model and the implementation?
- How can these three/four descriptions be maintained over time (e.g. 20 years for planes)?

Program Versus Language Based Abstraction

- Abstraction soundness is difficult to prove (undecidable);
- Difficulty is about the same whether it is program-based or language-based;
- Program-based abstractions are hardly reusable and highly sensible to program modifications;

Standard Abstractions for Program Analysis

- In static program analysis abstraction is language-based;
- The model of the implementation is provided by the analyzer and proved correct for a given programming language;
- Standard abstractions can be shared in the form of reusable libraries;
- Difficulty: since analyzers must work for infinitely many programs, no finite abstraction will be as powerful as infinite abstract domains (with widening/narrowing);

A few challenges:

- A broader class of general-purpose abstractions , implemented in the form of libraries, is needed;
- The problem of tailoring such abstractions to program-specific verification is:
  - Partly solved only from a theoretical point of view (abstract domain refinement);
  - Undecidable hence still opened in practice.

Widening/Narrowing and Their Duals

- Necessary to speed up fixpoint computations in infinite abstract domains;
- Widening/narrowing decide upon the abstraction during the verification process, not before;
- The success of program analyzers often relies on the design of subtle widenings/narrowings providing a good balance between cost and precision;
- Challenge: dual widening/narrowing (for approximation from below);

Specifications

Specifications in Model Checking

- User provided: temporal logic or fixpoint calculus;
- Challenges:
  - infinite past/future specifications (for which set of states based abstractions are incomplete);
  - make such specifications understandable and reusable;

Specifications in Program Analysis

- Provided automatically: absence of runtime errors, good programming practice [2];
- User provided: forward/backward and least/greatest fixpoints based static checking [3];
- Challenges: make specifications and static program analysis follow the program development process;

---

[2] threads must eventually enter/exit critical sections, the condition in monitors will eventually be verified for condition variables, etc.

[3] Very similar to linear temporal logic

Unbounded Control Structures

- Transitions systems are suitable for flat control structures (e.g. Prolog, procedureless C);
- Programming languages often involve unbounded control structures (recursion/reentrant software, process creation, race conditions with dynamic priorities, etc.);
- Challenge: precise abstractions of unbounded control structures.

Control Structures

Numerical Properties

Integer Properties

- Initial work in program analysis (e.g. convex polyhedral abstraction) reused in model-checking (e.g. of hybrid automata);
- Most work on linear safety properties;
- Challenges:
  - Little work on liveness properties with fairness hypotheses (generation of variant functions);
  - Little work on non-linear boundedness;

Floating Point Properties

- Very important in embedded software (e.g. to control trajectories);
- Evolution from fixed-point to floating-point computations;
- Difficulties:
  - run-time errors ,
  - cumulated loss of precision;
- Challenges:
  - Estimate and find the origin of uncontrolled loss of precision of the floating-point operations (without analysis-time errors/loss of precision!).

Data Structures

Data Structures

- Even trivial data structures can be a problem (e.g. type casts, buffer overflows);
- Low-level programming languages (C, Ada) used in embedded software make use of pointers (not even speaking of heap allocation e.g. in parameter passing);
- Challenge: pointer/alias analysis (hundreds of published papers but no cost/precision adjustable pointer analysis is presently emerging);

Modularity

Modular Program Analysis Techniques

- Simplification-based separate analysis;
- Worst-case separate analysis;
- Separate analysis with (user-provided) module interfaces;
- Symbolic lazy relational separate analysis;
- Iterated composition of the above separate local analyses and global analysis methods.
- Challenge:
  - Scale-up without precision loss and overwelming user interaction;

Timing

Timing

- Timing constraints are central to process control software;
- Timing constraints must be checked at the lowest machine level;
- Much progress has been done recently in static WCET estimation [4];
- Challenges:
  - Formalize the semantics of modern processors;
  - Design WCET analyzers parameterized by the processor semantics;

---

[4] see the presentation by R. Wilhelm and the demo by S. Thesing in this workshop.

Termination and Unbounded Liveness Properties

Finiteness Hypotheses

- Finiteness: every liveness property can be proved by proving a stronger safety property;
- Infiniteness: not always possible, so (transfinite) variant functions are required;
- Challenge: after understanding variant functions as abstractions, infer them automatically.

Fairness

- Solved problem for finite systems (fair model checking);
- Very difficult to find effective abstractions for infinite systems;
- Challenge:
  - Take scheduling into account (e.g. to statically detect potential priority inversions).

Distribution and Mobility

Network Integrated Embedded Software

- Critical real-time embedded software evolves from centralized to distributed control (modern automotive, aeronautic and train transportation computer systems certainly contain several dozen of computers communicating on a LAN);
- Predictable evolution towards integration into WANs (e.g. air traffic control) with continuously evolving communication topologies;
- More intelligent communication protocoles will certainly require mobile code;
- Challenge: scale up static analysis of distributed/mobile code;

User Interaction

- All formal methods ultimately require user interaction;
- Automatic program analysis often hard to understand (e.g. polymorphic type systems with subtypes);
- Challenges:
  - educate programmers on formal methods;
  - communicate/acquire complex reasonings about programs to/from users (not just counter-examples).

User Interfaces

DAEDALUS

Partners of DAƎDⱯLUS

- P. Cousot (ENS, France), scientific coordinator;
- R. Cousot (École polytechnique, France);
- A. Deutsch & D. Pilaud (PolySpace Technologies, France);
- C. Ferdinand (AbsInt, Germany);
- É. Goubault (CEA, France);
- N. Jones (DIKU, Denmark);
- F. Randimbivololona (Airbus, France), coord.;
- M. Sagiv (Univ. Tel Aviv, Israel);
- H. Seidel (Univ. Trier, Germany);
- R. Wilhelm (Univ. Sarrebrücken, Germany);

Long Term Investment

- Formal verification of embedded software is a challenge for the next decade;
- Program-based hand-made abstraction is extremely costly to design;
- Language-based hand-made abstraction is extremely costly to design but reusable;
- Therefore program analysis is an economically viable complement/alternative to model checking/deductive methods;
- Program analyzers are hard to design and implement (>>> compilers);

Conclusion

- Challenge: find support for the required long term intellectual investment.