Introduction

Static Analysis by Abstract Interpretation

Introduction

Xavier Rival

East China Normal University

November, 25th. 2013
Lecture main points

- Foundations of **static analysis** by **abstract interpretation**
- A static analyzer for the verification of **safety-critical embedded softwares**
- **Applications**

**This introduction:**
issues in **program verification** (for **embedded systems**)

An example: Ariane 5, flight 501

- European satellite launcher
- First flight on 4th of June 1996: failure
- Root cause:
  - 64 bits float to 16 bits integer overflow
  - loss of both inertial reference systems (same software, same bug!)
  - invalid data fed into the trajectory control computer
  - loss of control and destruction...
- Additional facts:
  - faulty computation useless after lift-off
  - software well-tested... with less powerful launcher

Full report: http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
Embedded systems verification

Common embedded softwares development techniques

- **Software qualification**
  - area specific *development rules*, e.g., DO 178 c in avionics
  - identification of *sources of risk*, use of *coding rules*
  - *code review*, etc

- **Use of high-level languages**
  - guarantee that some classes of errors should not happen
  - still, languages like C / C++ are very commonly used

- **Testing** by simulation
  - exhaustive simulation usually impossible

We need *formal methods* to *guarantee correctness properties*
Correctness properties to verify

- Absence of runtime errors or undefined behaviors (Astrée)
- User-specified invariance properties (Astrée)
- Non exhaustion of resources
- Termination
- Timing properties: a piece of code should execute in at most $t$ seconds
- ...

Two important categories of properties: safety and liveness

- Safety: some (bad) event will never happen
- Liveness: some (good) event will eventually happen

In this lecture we focus mainly on safety
Outline

1. Embedded systems verification

2. Verification techniques
   - Indecidability
   - Partial approaches to verification

3. Course overview
The termination problem

Termination

Program $P$ terminates on input $X$ if and only if any execution of $P$, with input $X$ eventually reaches a final state.

- **Final state:** final point in the program (i.e., not error)
- **We may want to ensure termination:**
  - processing of a task, such as, e.g., printing a document
  - computation of a mathematical function
- **We may want to ensure non-termination:**
  - operating system
  - device drivers

The termination problem

Can we find a program $P_t$ that takes as argument a program $P$ and data $X$ and that returns "TRUE" if $P$ terminates on $X$ and "FALSE" otherwise?
The termination problem is not computable

- **Proof by reductio ad absurdum**, using a *diagonal argument*
  We assume *there exists a program* $P_a$ *such that*:
  - $P_a$ always terminates
  - $P_a(P, X) = 1$ if $P$ terminates on input $X$
  - $P_a(P, X) = 0$ if $P$ does not terminate on input $X$

- We consider the following program:

```c
void P0(P){
    if(Pa(P, P) == 1){
        while(TRUE){}    //loop forever
    }else{
        return;    //do nothing...
    }
}
```

- **What is the return value of** $P_a(P_0, P_0)$?  
  i.e., $P_0$ does it terminate on input $P_0$?
The termination problem is not computable

- **What is the return value of** $P_a(P_0, P_0)$?
  
  We know $P_a$ always terminates and returns either 0 or 1 (assumption).
  Therefore, we need to consider only two cases:
  
  - if $P_a(P_0, P_0)$ returns 1, then $P_0(P_0)$ loops forever, thus $P_a(P_0, P_0)$ should return 0, so we have reached a **contradiction**
  - if $P_a(P_0, P_0)$ returns 0, then $P_0(P_0)$ terminates, thus $P_a(P_0, P_0)$ should 1, so we have reached a **contradiction**

  - In both cases, we reach a contradiction
  - Therefore we conclude no such a $P_a$ exists

The termination problem is not decidable

There exists no program $P_t$ that always terminates and always recognizes whether a program $P$ terminates on input $X$
Reduction to the termination problem

- Can we find a program $P_c$ that takes a program $P$ and input $X$ as arguments, always terminates and returns
  - 1 if and only if $P$ runs safely on input $X$, i.e., without a runtime error
  - 0 if $P$ crashes on input $X$

- Answer: No, the same diagonal argument applies

Non-computability result
The absence of runtime errors is not computable
**Rice theorem**

- **Semantic specification**: set of *correct* program executions
- **“Trivial” specifications**:
  - empty set
  - set of all possible executions
- ⇒ intuitively, the non interesting verification problems...

**Rice theorem (1953)**

*Considering a Turing complete language, any non trivial specification is not computable*

- **Intuition**: there is no algorithm to decide non trivial specifications, starting with only the program code
- Therefore all interesting properties are *not computable*:
  - termination,
  - absence of runtime errors,
  - absence of arithmetic errors, etc...
Outline

1. Embedded systems verification

2. Verification techniques
   - Indecidability
   - Partial approaches to verification

3. Course overview
Towards partial solutions

- The initial verification problem is not computable
- **Solution:** solve a weaker problem
- **Several compromises can be made:**
  - **simulation / testing:** observe only finitely many finite executions
    infinite system, but only finite exploration (no proof beyond that)
  - **assisted theorem proving:** we give up on automation
    (no proof inference algorithm in general)
  - **model checking:** we consider only finite systems
    (with finitely many states)
  - **bug-finding:** search for “patterns” indicating “likely errors”
    (may miss real program errors, and report non existing issues)
  - **static analysis with abstraction:** attempt at automatic correctness proofs
    (yet, may fail to verify some correct programs)
Verification techniques

Partial approaches to verification

Safety verification method characteristics

Safety verification problem

- **Semantics** \([P]\) of program \(P\): set of behaviors of \(P\) (e.g., states)
- **Property to verify** \(S\): set of admissible behaviors (e.g., safe states)

- **Automation**: existence of an algorithm
- **Scalability**: should allow to handle large softwares
- **Soundness**: identify any wrong program
- **Completeness**: accept all correct programs
- **Apply to program source code**, i.e., not require a **modelling phase**
Testing by simulation

- **Principle**: run the program on finitely many finite inputs
- **Very widely used**:
  - **unit testing**: each function is tested separately
  - **integration testing**: with all surrounding systems, hardware e.g., iron bird in avionics
- **Automated**
- **Complete**: will never raise a false alarm
- **Unsound** unless exhaustive: may miss program defects
- **Costly**: needs to be re-done when software gets updated
Principle: have a machine checked proof, that is partly human written
  ▶ tactics / solvers may help in the inference
  ▶ the hardest invariants have to be user-supplied

Applications
  ▶ industry (rare): Line 14 in Paris Subway
  ▶ hardware: ACL 2
  ▶ accademia: CompCert compiler, SEL4 verified micro-kernel

Not fully automated
  often turns out costly as complex proof arguments have to be found

Sound and complete
Model-Checking

- **Principle**: consider *finite systems*
  - many algorithms for *exhaustive exploration, symmetry reduction*...

- **Applications**:
  - *hardware* verification
  - *driver protocols* verification (Microsoft)

- Applies on a *model*: a model extraction phase is needed
  - for infinite systems, this is *necessarily approximate*
  - not always automated

- **Automated, sound, complete** with respect to the model
“Bug finding”

- **Principle**: identify “likely” issues
- **Example**: Coverity
- **Automated**
- **Not complete**: may report false alarms
- **Not sound**: may accept false programs
  thus inadequate for safety-critical systems
Use some approximation, but always in a conservative manner

- **Under-approximation** of the property to verify: $S_{\text{under}} \subseteq S$
- **Over-approximation** of the semantics: $\llbracket P \rrbracket \subseteq \llbracket P \rrbracket_{\text{upper}}$
- We let an automatic static analyzer attempt to prove that:
  $$\llbracket P \rrbracket_{\text{upper}} \subseteq S_{\text{under}}$$

If it succeeds, $\llbracket P \rrbracket \subseteq S$

- In practice, the static analyzer computes $\llbracket P \rrbracket_{\text{upper}}, S_{\text{under}}$
  (computable)
Static analysis with abstraction (2/4)

Soundness
The abstraction will catch any incorrect program

- If $[P] \not\subseteq S$, then $[P]_{\text{upper}} \not\subseteq S_{\text{under}}$

since
\[
\begin{cases}
S_{\text{under}} \subseteq S \\
[P] \subseteq [P]_{\text{upper}}
\end{cases}
\]
**Static analysis with abstraction (3/4)**

**Incompleteness**

The abstraction may fail to certify **some correct programs**

**Case of a false alarm:**

- program $P$ is **correct**
- but the static analysis **fails**
Incompleteness

The abstraction may fail to certify some correct programs

- In the following case, the analysis cannot conclude anything

- One goal of the static analyzer designer is to avoid such cases

Static analysis using abstraction

- **Automatic**: $[P]_{\text{upper}}$, $S_{\text{under}}$ computed automatically
- **Sound**: reports any incorrect program
- **Incomplete**: may reject correct programs
Outline

1. Embedded systems verification
2. Verification techniques
3. Course overview
Foundations of Abstract Interpretation

- theory of order relations
- abstraction relation
- computation of sound abstract semantics and widening
- design of a simple abstract interpreter
Course overview

Abstract domains

- numerical abstract domains
  - intervals
  - octagons
  - polyedra
  - ellipsoids...

- symbolic abstract domains
  - boolean relations
  - trace partitioning
Use of the Astrée static analyzer

- demonstration: interface, analysis stages
- lab sessions, following the lectures on the main abstract domains
- analysis of example applications in the last lab sessions
Applications of abstract interpretation

- **Scalable numerical abstract domains**
  (Medhi Bouaziz)

- **An abstract domain to infer types of zones in spreadsheets**
  (Tie Cheng)

- **Internal coarse-graining of molecular systems**
  (Jérôme Feret)

- **Static analysis of asynchronous softwares**
  (Antoine Miné)

- **Modular construction of shape-numeric analyzers**
  (Xavier Rival)

- **The abstract domain of piecewise-defined ranking functions**
  (Caterina Urban)