Introduction to Software Verification

Orna Grumberg

Lectures Material
winter 2015-16
Lecture 1
Why (formal) Verification?

- Safety-critical applications:
  - Air-traffic controllers
  - Medical equipment
  - Cars

  Bugs are unacceptable!

- Bugs found in later stages of design are expensive, e.g. Intel’s Pentium bug in floating-point division

- Testing does not provide full coverage
What are we doing about it?

Testing – build it, try it on some cases, hope it works for all cases.
What should we be doing?

Formal analysis and verification
The goal of the course: Formal Verification

Given

- A (model of) hardware or software system and
- a formal specification

does the system satisfy the specification?

Not decidable!
Formal Verification

Solutions:

- "Program correctness":
  Provide non-automated verification methods

- "Automatic verification / Model Checking":
  restrict the problem to a decidable one:
  - Finite-state reactive systems
  - Propositional temporal logics
Specifications

• Should be given for a system by the designer, developer, programmer, user

• Examples:
  - Does the program always terminate?
  - Does the program compute correctly multiplication of its inputs?
Specifications

• Additional examples:
  - When we press a sequence of buttons on the control panel of an airplane / microwave - do we get the desired result?
  - When we deposit money - does it get to our account?
  - Can a user access data only if he has the appropriate authorization?
Verification tools

Are developed and used in

• **Hardware industry:** Intel, IBM, Cadence, Mellanox, ...

• **Software industry:** Microsoft, NASA, ...

• **Universities**
Part 1 of the course

Program Correctness

- Non-automated
- Verify program with possibly infinite number of states
- Refer to the programs as input-output transformation
Ingredients for Formal Verification

1. Specification language
   • With formal semantics

2. Programming language
   • with formal semantics

3. Proof rules
   • For proving “Program P has the property $\phi$”
Requirements from the proof rules

- **Soundness of the rules**: if we were able to prove correctness of program \( P \) w.r.t. specification \( \varphi \) using the proof rules, then \( P \) is correct w.r.t. \( \varphi \)

- **Completeness of the rules**: if \( P \) is correct w.r.t. specification \( \varphi \), then our proof rules can prove it
We handle:

• **Deterministic programs**
  - Exactly *one computation for every input*
  - *At most one output for each input*

• **Properties**
  - *Partial correctness*
  - *Termination*
  - *Total Correctness*
Some notations

- **Program variables**: $\bar{x} = (x_1, \ldots, x_n)$

- **A state of the program** $\sigma$ **is a function from program variables to their domains**

- **The set of program states is defined by**: 
  $$D_1 \times \ldots \times D_n \cup \{\bot\}$$ 
  Where $D_i$ is the domain of variable $x_i$
Program states: Examples

- A program with integer variable $x$, Boolean variable $b$
  - States: $(5, F), (-17, T)$

- Elevator on 3 floors:
  - on_floor1, on_floor2, on_floor3: Boolean
  - in_elev1, in_elev2, in_elev3: Boolean
  - direction $\in \{\text{up, down}\}$, door $\in \{\text{open, close}\}$
  - State: $(F, T, T, T, T, F, \text{up, close})$
Defining the Specification

**Specification** is a pair \(<q_1(\bar{x}), q_2(\bar{x})>\)

where:

- \(q_1(\bar{x}), q_2(\bar{x})\) are first order formulas over program variables

- \(q_1(\bar{x})\) describes a condition holding **before** the execution of the program

- \(q_2(\bar{x})\) describes a condition holding **at the end** of the execution of the program
Examples

Specification example

• \( < (x \geq 0 \land y > 0), (z = x/y \land z \geq 0) > \)

A program with \( x \in \mathbb{N}, y \in \mathbb{R}, b \in \{T,F\} \)
States: \((5, 5.0, T), (7, 3.111, F)\)

\( q_1(x, y, b) = x > 0 \land b \)

\( q_2(x, y, b) = x+y > 0 \land \neg b \)
Computations of Programs

• \( \pi(P, \sigma) \) denotes a computation of program \( P \) from state \( \sigma \)

• \( \pi(P, \sigma) \) is a finite \((\sigma_1, \ldots, \sigma_k)\) or infinite \((\sigma_1, \sigma_2, \ldots)\) sequence of states where:
  - \( \sigma_1 = \sigma \)
  - \( \sigma_{i+1} \) is a result of applying an action from the program on \( \sigma_i \)

• This definition is not a full definition
More notations

• \( \perp \) - bottom : the undefined value

• \( \text{val}(\pi) \) denotes the final state of computation \( \pi \) (if exists)
  - \( \text{val}(\pi) = \sigma_k \) if \( \pi = (\sigma_1, \ldots, \sigma_k) \)
  - \( \text{val}(\pi) = \perp \) if \( \pi = (\sigma_1, \sigma_2, \ldots) \)
    - \( \pi \) is an infinite computation

• \( \sigma \models q(\overline{x}) \) if \( q(\overline{x}) \) is true when free variables in \( q \) are replaced with matching values in \( \sigma \)
• **Important remark:**

\[ \bot \not\equiv q(\overline{x}) \text{ for every } q(\overline{x}) \text{ (even } \bot \not\equiv \text{ true)} \]

• **Example of formulas and their meaning:**

\[ q(y) = \forall x(y|x \lor 2\neg x) \text{ where } x,y \text{ are naturals} \]

- For a state \( \sigma (x) = 1, \sigma (y) = 2, \sigma (z) = 1 \)

\[ \sigma \models q(y) \text{ since } \forall x(2|x \lor 2\neg x) \text{ is true} \]
Partial Correctness

- A program $P$ is partially correct with respect to specification $<q_1(\bar{x}), q_2(\bar{x})>$ iff for every computation $\pi$ of $P$ from an initial point of $P$, and for every state $\sigma_0$:
  - the computation starts from state $\sigma_0$ which satisfies $q_1(\bar{x})$ and
  - the computation terminates

  then
  - $q_2(\bar{x})$ holds at the end of the computation
Partial Correctness

• For every computation $\pi$ and every state $\sigma_0$:

\[
(\sigma_0 \models q_1(\bar{x}) \text{ and } \text{val}(\pi(P, \sigma_0)) \neq \perp) \Rightarrow
\text{val}(\pi(P, \sigma_0)) \models q_2(\bar{x})
\]

• Notation: $\{q_1\}P\{q_2\}$
Total Correctness

- A program $P$ is *totally correct* with respect to specification $<q_1(x), q_2(x)>$ iff for *every* computation $\pi$ of $P$ from an initial point of $P$, and for *every* state $\sigma_0$:
  
  if
  
  - the computation starts from state $\sigma_0$ which satisfies $q_1(x)$
  then
  
  - the computation *terminates*, and
  
  - $q_2(x)$ holds at the end of the computation
Total Correctness

• For every computation $\pi$ and every state $\sigma_0$:

\[
\sigma_0 \models q_1(\bar{x}) \Rightarrow \text{val}(\pi(P, \sigma_0)) \neq \bot \quad \text{and} \quad \text{val}(\pi(P, \sigma_0)) \models q_2(\bar{x})
\]

• Notation: $<q_1>P<q_2>$
How do we write the specification:

“$P$ terminates if the initial state satisfies $q_1$“
Separation Lemma

• For every program $P$ and specification $<q_1,q_2>$:
  
  $\models <q_1> P <q_2>$
  
  if and only if
  
  $\models \{q_1\} P \{q_2\}$ and $\models <q_1> P \langle \text{true} \rangle$
Examples

• Which programs satisfy \{true\}P\{false\}?

• Which programs satisfy \langle true\rangle P\langle false\rangle?
Logical Variables in Specifications

Example 1:
Specify a program with a single variable $x$ whose value at the end of the computation is twice its value at the beginning.
Logical Variables in Specifications

Solution: add fresh variables which are
- not part of the program and therefore
- their value does not change during the execution of the program

These variables are called logical variables

Con convention: We use logical variable $X$ to preserve the value of variable $x$
Example 2:
Program which returns in variable \( z \) the multiplication of variables \( x \) and \( y \)

Convention:
Assertions \( q_1, q_2 \) are now defined over \( \bar{x} \) that includes both program variables and logical variables
While Programs: Syntax

\[ S ::= x := e \mid \text{skip} \mid S_1 ; S_2 \mid \]
\[ \quad \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \mid \]
\[ \quad \text{while } B \text{ do } S \text{ od} \]

- **B** - condition - predicate over program variables
- **e** - expression over program variables
- **skip, x := e** - atomic actions
- **The rest** - compound actions
Operational semantics of while Programs

• $\sigma$ - program state
• Configuration of a program is a tuple $C = \langle S, \sigma \rangle$ such that $S$ is a while program and $\sigma$ is a state
• The program $E$ is the empty program, and for every program $S$:
  $E;S = S;E = S$
• A configuration is halting if $S=E$
Relation $\rightarrow$ Over Configurations

$\rightarrow$ is the smallest relation that satisfies:

1. $<x:=e, \sigma> \rightarrow <E, \sigma[x \leftarrow \sigma(e)]>

2. $<\text{skip}, \sigma> \rightarrow <E, \sigma>

3. For every while-program $T$:
   
   if $<S, \sigma> \rightarrow <S', \sigma'>$
   
   then $<S; T, \sigma> \rightarrow <S'; T, \sigma'>$
Relation → Over Configurations

4. If $\sigma(B)=true$ ($\sigma\models B$) then
   $<$if $B$ then $S_1$ else $S_2$ fi, $\sigma$$>$ → $<$S$_1$, $\sigma$$>$
else ($\sigma(B)$=false)
   $<$if $B$ then $S_1$ else $S_2$ fi, $\sigma$$>$ → $<$S$_2$, $\sigma$$>$

5. If $\sigma(B)$=true then
   $<$while $B$ do $S$ od, $\sigma$$>$ → $<$S;while $B$ do $S$ od, $\sigma$$>$
else ($\sigma(B)$=false)
   $<$while $B$ do $S$ od, $\sigma$$>$ → $<$E, $\sigma$$>$