IN5170/IN9170 – Models of concurrency

Fall 2021

Compiled: September 22, 2021

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [**?**]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

Contents

1 Program Analysis Part I : Sequential Programs (week 4)

15. 9. 2021

Program correctness

Is my program correct?

Central question for this lecture (Part I) and Part II.

- Does a given program behave as intended?
- Surprising behavior?

$$x := 5; \{\ x = 5\ \}\langle x := x + 1\rangle; \{\ x =?\ \}$$

- clear: $x = 5$ *immediately* after first assignment
- Will this still hold when the second assignment is executed?
  - Depends on other processes $\Rightarrow$ **interference**
- What will be the final value of x?

Today (Part I): Basic machinery for sequential program reasoning (Part II): Extending this machinery to the concurrent setting

Verification of global invariants

To prove a global invariant I, we must prove:

$$\{I\}\ \mathtt{s}\ \{I\}$$

for each atomic statement/critical region s.

Example 1 (Consider the await language:).

- $\{I\} < \mathtt{s} > \{I\}$ reduces to $\{I\}\ \mathtt{s}\ \{I\}$

  Note: This is a sequential verification problem!

- $\{I\} <$ **await** $\mathtt{b}; \mathtt{s} > \{I\}$ reduces to $\{I \wedge b\}\ \mathtt{s}\ \{I\}$

  Note: Again a sequential verification problem!

Reasoning about semaphores and signalling is more complex. So is local reasoning and interference. Next time.

Concurrent executions

- Concurrent program: several threads operating on (here) *shared* variables
- Parallel updates to x and y:

$$\textbf{co}\ \langle x := x * 3; \rangle \parallel \langle y := y * 2; \rangle\ \textbf{oc}$$

- Every (concurrent) execution can be written as a sequence of atomic operations (gives one history)
- Two possible histories for the above program
- Generally, if n processes executes m atomic operations each:

$$\frac{(n * m)!}{m!^n} \qquad \text{If n=3 and m=4:} \frac{(3 * 4)!}{4!^3} = 34650$$

How to verify program properties?

- *Testing* or *debugging* increases confidence in the program correctness, but does not guarantee *correctness*
  - Program testing can be an effective way to show the presence of bugs, but not their absence

- *Operational reasoning* (exhaustive case analysis) tries all possible executions of a program, not so practical for concurrent programs

- *Formal analysis* (assertional reasoning) allows to *deduce* the correctness of a program without executing it
  - Specification of program behavior
  - Formal argument that the specification is correct
    - Verification
    - (Symbolic) Model Checking

1.1 Specification

Formal Specifications

Kinds of formal specifications:

- pre/post conditions
- loop invariant
- class invariants. monitor invariants, data invariants
- global and local invariants in concurrent systems
  - may be violated inside critical regions

Formal specifications are useful

- when designing a system, using the specification as a guide
- in order to understand a program
- using the specification as a contract between different units,
  - in particular concurrent units
- for testing, using specifications in testing
  - for test case generation

States

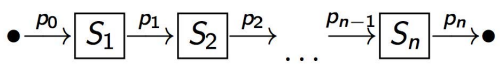
- *state* of a program consists of the values of the program variables at a point in time, example: $\{\ x = 2 \wedge y = 3\ \}$
- The state space of a program is given by the different values that the declared variables can take
- Sequential program: one execution thread operates on its own state space
- The state may be *changed* by assignments (in "imperative" programming)

Example 2.

$$\{\ x = 5 \wedge y = 5\ \}\texttt{x := x * 2;}\{\ x = 10 \wedge y = 5\ \}\texttt{y := y * 2;}\{\ x = 10 \wedge y = 10\ \}$$

Executions

- Given program S as sequence $S_1; S_2; \ldots; S_n;$, starting in a state p_0:



  where $p_1, p_2, \ldots p_n$ are the different states during execution

- Can be documented by: $\{p_0\}S_1\{p_1\}S_2\{p_2\}\ldots\{p_{n-1}\}S_n\{p_n\}$
- p_0, p_n gives an external specification of the program: $\{p_0\}S\{p_n\}$
- We often refer to p_0 as the *initial* state and p_n as the *final* state

Example 3 (from previous slide).

$$\{\ x = 5 \wedge y = 5\ \}\ \mathtt{x} := \mathtt{x} * 2;\ \mathtt{y} := \mathtt{y} * 2;\ \{\ x = 10 \wedge y = 10\ \}$$

Assertions

Want to express more general properties of programs, like invariants:

$$\{\ x = y\ \}\ \mathtt{x} := \mathtt{x} * 2; \mathtt{y} := \mathtt{y} * 2;\ \{\ x = y\ \}$$

- If the assertion $x = y$ holds, when the program *starts*, $x = y$ will also hold when/if the program *terminates*
- Does not talk about specific, concrete *values* of x and y, but about *relations* between their values
- Assertions characterize sets of states

Example 4. The assertion $x = y$ describes *all* states where the values of x and y are equal, like $\{x = -1 \wedge y = -1\}$, $\{x = 1 \wedge y = 1\}$, ...

Assertions

state assertion P: defines the *set* of states where P is true

Example 5.

$x = y$	all states where x has same value as y
$x \leq y$:	all states where the value of x is less or equal to the value of y
$x{=}2 \wedge y{=}3$	the state where x is 2 and y is 3
$x{=}2 \vee y{=}3$	the state where x is 2 or y is 3
$x{=}2 \Rightarrow y{=}3$	the state where y is 3 if x is 2
true	all states
false	no state

Example 6.

$$\{\ x = y\ \}\mathtt{x} := \mathtt{x} * 2;\{\ x = 2 * y\ \}\mathtt{y} := \mathtt{y} * 2;\{\ x = y\ \}$$

Assertions may or may not say something *correct* for the behavior of a program. In this example, the assertions say something correct.

Formal analysis of programs

- Establish program properties/correctness, using a system for formal reasoning
- Help in understanding how a program behaves
- Useful for program construction
  - in particular, class and loop invariants
- Look at logics for formal analysis
- basis of analysis tools

1.2 Formal System

Formal system

- *Axioms:* Defines the meaning of individual program statements
- *Rules:* Derive the meaning of a program from the individual statements in the program

Logics and formal systems

Our formal system consists of:

- a language, formed by syntactic building blocks:
  - A set of *symbols* (constants, variables,...)
  - A set of *formulas* (meaningful combination of symbols)
- derivation machinery
  - A set of *axioms* (assumed to be true) [1]
  - A set of *inference rules*

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

  - H_i: *premise* (sometimes called *assumption*),
  - C : *conclusion*
  - intention: conclusion is true if all the premises are true
  - The inference rules specify how to derive new formulas from axioms and already derived formulas.
- a semantics, defining the meaning of the formulas

1.3 First-Order Logic

Symbols (first-order logic with equality)

- variables: $x, y, z, \ldots$ which include
  - program variables
  - + "extra" ones including logical variables
- Function symbols: $+, -, \ldots$, and constants $0, 1, 2, \ldots, \mathit{true}, \mathit{false}$
- Equality (a predefined relation symbol): $=$
- Other relation symbols: $\leq, \geq, \ldots$
- Connectives $\neg, \vee, \wedge, \Rightarrow$
- Quantifiers: $\forall, \exists$

Formulas of first-order logic

Meaningful combination of symbols

Assume that A and B are formulas, then the following are also formulas:

$\neg A$	means "not A"
$A \vee B$	means "A or B"
$A \wedge B$	means "A and B"
$A \Rightarrow B$	means "A implies B"

If x is a variable and A a formula, the following are formulas:[2]

$\forall x : A(x)$	means "A is true for all values of x"
$\exists x : A(x)$	means "there is (at least) one value of x such that A is true"

[1] **NB:** An axiom can be seen as a rule with no premise.
[2] $A(x)$ to indicate that, here, A (typically) contains x.

Examples of axioms and rules (propositional logic)

Typical axioms:

$$A \vee \neg A \qquad\qquad A \Rightarrow A \qquad\qquad \mathit{true}$$

Typical rules:[3]

$$\frac{A \quad B}{A \wedge B}\ \textsc{And-I} \qquad \frac{A}{A \vee B}\ \textsc{Or-I} \qquad \frac{A \Rightarrow B \quad A}{B}\ \textsc{Impl-E/modus ponens}$$

Example 7.

$$\frac{x = 5 \quad y = 5}{x = 5 \wedge y = 5}\ \textsc{And-I} \qquad \frac{x = 5}{x = 5 \vee y = 5}\ \textsc{Or-I}$$

$$\frac{x \geq 0 \Rightarrow y \geq 0 \quad x \geq 0}{y \geq 0}\ \textsc{Impl-E}$$

Important terms

- **Proof**: derivation tree where all leaf nodes are axioms
- **Theorems**: a "formula" derivable in a given proof system

- **Interpretation**: defines each formula as either *semantically true* or *false*. Semantically true is called *valid*.
- **Soundness** (of the logic): If we can prove ("derive") some formula P (in the logic) then P is actually semantically true
- **Completeness**: If a formula P is semantically true, then it can be proved

soundness and completeness means:

semantically true formulas = provable formulas

or

validity = provability

or

$\models = \vdash$

1.4 Hoare Logic

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are of the form

  "Hoare triple"

  $$\{\ P_1\ \}\ S\ \{\ P_2\ \}$$

  - S: program statement (or sequence of statements)
  - P, P_1, P', Q ...: assertions over program states
    - including use of $\neg, \wedge, \vee, \Rightarrow, \exists, \forall$
  - In above triple: P_1 pre-condition, and P_2 post-condition of S

Example 8.

$$\{\ x = y\ \}\ \texttt{x := x * 2; y := y * 2;}\ \{\ x = y\ \}$$

[?, ?, ?]

[3] The "names" of the rules are written on the right of the rule, they serve for "identification". By some convention, "I" stands for rules *introducing* some logical connector, "E" for *eliminating* one.

The proof system PL (Hoare logic)

- Express and prove program properties
$$\{P\}\ S\ \{Q\}$$
  - (P, Q) may be seen as a specification of the program S, sometimes called a contract
  - Code analysis by proving the specification (in PL)
  - Symbolic and static analysis:
    - No need to execute the code in order to do the analysis
  - An interpretation maps triples to *true* or *false*
    - $\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 1\ \}$ should be semantically *true*
    - $\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 0\ \}$ should be semantically *false*

Reasoning about programs

- Basic idea: *Specify* what the program is supposed to do (pre- and post-conditions)
- Pre- and post-conditions are given as assertions over the program state
- use PL for a mathematical argument that the program satisfies its specification

Interpretation:
Interpretation ("semantics") of triples is related to program execution

Partial correctness interpretation
$\{\ P\ \}\ S\ \{\ Q\ \}$ is valid, denoted $\models \{\ P\ \}\ S\ \{\ Q\ \}$:

- If the initial state of S satisfies P (P holds for the initial state of S) and
- **if**[4] S *terminates,*
- *then* Q is *true* in the final state of S

Expresses *partial correctness* (termination of S is assumed)

Example 9. $\{x = y\}$ `x := x * 2; y := y * 2;` $\{x = y\}$ is *true* if the initial state satisfies $x = y$ and if the execution terminates, then the final state satisfies $x = y$

Examples
Some valid triples (i.e., semantically true)

$$\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 1\ \}$$
$$\{\ x = 4\ \}\ x := 5;\ \{\ x = 5\ \}$$
$$\{\ \textbf{true}\ \}\ x := 5;\ \{\ x = 5\ \}$$
$$\{\ y = 4\ \}\ x := 5;\ \{\ y = 4\ \}$$
$$\{\ x = 4\ \}\ x := x + 1;\ \{\ x = 5\ \}$$
$$\{\ x = a \wedge y = b\ \}\ x := x + y;\ \{\ x = a + b \wedge y = b\ \}$$
$$\{\ x = 4 \wedge y = 7\ \}\ x := x + 1;\ \{\ x = 5 \wedge y = 7\ \}$$
$$\{\ x = y\ \}\ x := x + 1; y := y + 1;\ \{\ x = y\ \}$$

Some unvalid triples (i.e., not semantically true)

$$\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 0\ \}$$
$$\{\ x = 4\ \}\ x := 5;\ \{\ x = 4\ \}$$
$$\{\ x = y\ \}\ x := x + 1; y := y - 1;\ \{\ x = y\ \}$$
$$\{\ x > y\ \}\ x := x + 1; y := y + 1;\ \{\ x < y\ \}$$

[4] Thus: if S does not terminate, all bets are off...

Partial correctness

- The interpretation of $\{ P \}\ S\ \{ Q \}$ assumes/ignores termination of S, since termination is not proven.
- The pre/post specification (P, Q) expresses *safety* properties

  The state assertion *true* can be viewed as all states. The assertion *false* can be viewed as no state. What does each of the following triple express?

$\{ P \}\ S\ \{ \mathit{true} \}$	trivially true
$\{ P \}\ S\ \{ \mathit{false} \}$	S does not terminate for prestate P
$\{ \mathit{true} \}\ S\ \{ \mathit{false} \}$	S does not terminate
$\{ \mathit{true} \}\ S\ \{ Q \}$	Q holds after S for all prestates (provided S terminates)
$\{ \mathit{false} \}\ S\ \{ Q \}$	trivially true

Can express non-termination, but not termination!

Proof system PL

A proof system consists of *axioms* and *rules*
here: structural analysis of programs

- Axioms for basic statements:
  - $x := e$, **skip**,...
- Rules for composed statements:
  - $S_1;S_2$, **if**, **while**, **await** , **co** . . . **oc**, . . .

Formulas in PL

- formulas = triples
- theorems = derivable formulas[5]
- hopefully: all derivable formulas are also "really" (= semantically) true
- derivation: starting from axioms, using derivation rules
- $$\frac{H_1 \quad H_2 \quad \ldots \quad H_n}{C}$$
- axioms: can be seen as rules without premises

Soundness $-\vdash$ **implies** $\models$

If a triple $\{ P \}\ S\ \{ Q \}$ is derivable in PL, the triple holds (is valid):

- Example: we want
  $$\{ x = 0 \}\ x := x + 1\ \{ x = 1 \}$$
  to be a theorem (since it was interpreted as *true*),
- but
  $$\{ x = 0 \}\ x := x + 1\ \{ x = 0 \}$$
  should *not* be a theorem (since it was interpreted as *false*)

Soundness:[6]
All theorems in PL hold

$$\vdash \{ P \}\ S\ \{ Q \} \quad \text{implies} \quad \models \{ P \}\ S\ \{ Q \} \tag{1}$$

If we can use PL to prove some property of a program, then this property will hold for all executions of the program.

[5] The terminology is standard from general logic. A "theorem" in an derivation system is a *derivable* formula. In an ill-defined (i.e., unsound) derivation or proof system, theorems may thus be not true.

[6] technically, we'd need a semantics for reference, otherwise it's difficult to say what a program "really" does.

Textual substitution

Substitution
$P[e/x]$ means, all free occurrences of x in P are replaced by expression e. (Notation of book: $P_{x \leftarrow e}$)

Example 10.

$$\begin{array}{lll} (x = 1)[(x+1)/x] & \Leftrightarrow & x + 1 = 1 \\ (x + y = a)[(y + x)/y] & \Leftrightarrow & x + (y + x) = a \\ (y = a)[(x + y)/x] & \Leftrightarrow & y = a \end{array}$$

Substitution propagates into formulas:

$$\begin{array}{lll} (\neg A)[e/x] & \Leftrightarrow & \neg(A[e/x]) \\ (A \wedge B)[e/x] & \Leftrightarrow & A[e/x] \wedge B[e/x] \\ (A \vee B)[e/x] & \Leftrightarrow & A[e/x] \vee B[e/x] \end{array}$$

Free and "non-free" variable occurrences
$P[e/x]$

- Only *free* occurrences of x are substituted
- Variable occurrences may be *bound* by quantifiers, then that occurrence of the variable is not free (but bound)

Example 11 (Substitution).

$$\begin{array}{lll} (\exists y : x + y > 0)[1/x] & \Leftrightarrow & \exists y : 1 + y > 0 \\ (\exists x : x + y > 0)[1/x] & \Leftrightarrow & \exists x : x + y > 0 \\ (\exists x : x + y > 0)[x/y] & \Leftrightarrow & \exists z : z + x > 0 \end{array}$$

Correspondingly for $\forall$

The assignment axiom – Motivation
Given by backward construction over the assignment:

- Given the postcondition to the assignment, we may derive the precondition!

What is the precondition?

$$\{\ ?\ \}\ x := e\ \{\ x = 5\ \}$$

If the assignment `x = e` should terminate in a state where `x` has the value 5, the expression `e` must have the value 5 before the assignment:

$$\begin{array}{rll} \{\ e = 5\ \} & \texttt{x := e} & \{\ x = 5\ \} \\ \{\ (x = 5)[e/x]\ \} & \texttt{x := e} & \{\ x = 5\ \} \end{array}$$

Axiom of assignment
"Backwards reasoning:" Given a postcondition, we may construct the precondition:

Axiom for the assignment statement

$$\{\ P[e/x]\ \}\ x := e\ \{\ P\ \} \quad \textsc{Assign}$$

If the assignment $x := e$ should lead to a state that satisfies P, the state before the assignment must satisfy P when x is replaced by e. **NB:** e evaluated in the prestate = value of x in the poststate

PL structural inference rules

$$\frac{P' \Rightarrow P \qquad \{\ P\ \}\ S\ \{\ Q\ \} \qquad Q \Rightarrow Q'}{\{\ P'\ \}\ S\ \{\ Q'\ \}}\ \textsc{Consequence}$$

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \qquad \{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P \wedge P'\ \}\ S\ \{\ Q \wedge Q'\ \}}\ \textsc{Conjunction}$$

$$\frac{\{\ P\ \}\ S_1\ \{\ R\ \} \qquad \{\ R\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ S_1; S_2\ \{\ Q\ \}}\ \textsc{Seq}$$

- Red: logical proof obligations

Axiom of skip

The skip statement does nothing

Axiom:

$$\{\ P\ \}\ \textbf{skip}\ \{\ P\ \}\quad \textsc{Skip-axiom}$$

Proving an assignment

To prove the triple $\{\ P\ \}x := e\{\ Q\ \}$ in PL, we must show that the precondition P implies $Q[e/x]$, using the consequence rule.

$$\frac{P \Rightarrow Q[e/x] \qquad \{\ Q[e/x]\ \}\ x := e\ \{\ Q\ \}}{\{\ P\ \}\ x := e\ \{\ Q\ \}}$$

The red implication is a logical proof obligation. In this course we will not prove proof obligations formally.

- $Q[e/x]$ is the largest set of states such that the assignment is guaranteed to terminate with Q
- largest set corresponds to weakest condition $\Rightarrow$ weakest-precondition reasoning

Examples

$$\frac{\mathit{true} \Rightarrow 1 = 1}{\{\ \mathit{true}\ \}\ x := 1\ \{\ x = 1\ \}}$$

$$\frac{x = 0 \Rightarrow x + 1 = 1}{\{\ x = 0\ \}\ x := x + 1\ \{\ x = 1\ \}}$$

$$\frac{(x = a \wedge y = b) \Rightarrow x + y = a + b \wedge y = b}{\{\ x = a \wedge y = b\ \}\ x := x + y\ \{\ x = a + b \wedge y = b\ \}}$$

$$\frac{x = a \Rightarrow 0 * y + x = a}{\{\ x = a\ \}\ q := 0\ \{\ q * y + x = a\ \}}$$

$$\frac{y > 0 \Rightarrow y \geq 0}{\{\ y > 0\ \}x := y\{\ x \geq 0\ \}}$$

PL inference rules

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad P \wedge \neg B \Rightarrow Q}{\{\ P\ \}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi}\ \{\ Q\ \}}\ \textsc{Cond1}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S'\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S'\ \{\ Q\ \}}\ \textsc{Cond2}$$

$$\frac{\{\ I \wedge B\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\ I \wedge \neg B\ \}}\ \textsc{While}$$

- Red: logical proof obligations
- the rule for **while** needs a *loop invariant*!
- **for**-loop: exercise 2.22!

Sequential composition and consequence

Backward construction over assignments:

$$\frac{x = y \Rightarrow 2x = 2y}{\{\ x = y\ \}\ x := 2x\ \{\ x = 2y\ \}} \qquad \{\ (x = y)[2y/y]\ \}\ y := 2y\ \{\ x = y\ \}$$
$$\{\ x = y\ \}\ x := 2x; y := 2y\ \{\ x = y\ \}$$

Sometimes we don't bother to write down the assignment axiom:

$$\frac{\dfrac{(q * y) + x = a \Rightarrow ((q + 1) * y) + x - y = a}{\{\ (q * y) + x = a\ \}\ x := x - y;\ \{\ ((q + 1) * y) + x = a\ \}}}{\{\ (q * y) + x = a\ \}\ x := x - y; q := q + 1\ \{\ (q * y) + x = a\ \}}$$

Logical variables

- Do *not* occur in program text
- Used only in *assertions*
- May be used to "freeze" initial values of variables
- May then talk about these values in the postcondition

Example 12.

$$\{\ x = x_0\ \}\ \textbf{if}\ (x < 0)\ \textbf{then}\ x := -x\ \textbf{fi}\ \{\ x \geq 0 \wedge (x = x_0 \vee x = -x_0)\ \}$$

where $(x = x_0 \vee x = -x_0)$ states that

- the final value of x equals the initial value, *or*
- the final value of x is the negation of the initial value

Example: if statement

Verification of:

$$\{\ x = x_0\ \}\ \textbf{if}\ (x < 0)\ \textbf{then}\ x := -x\ \textbf{fi}\ \{\ x \geq 0 \wedge (x = x_0 \vee x = -x_0)\ \}$$

$$\frac{\{P \wedge B\}\ S\ \{Q\} \qquad (P \wedge \neg B) \Rightarrow Q}{\{\ P\ \}\ \textbf{if}\ \ B\ \textbf{then}\ \ S\ \textbf{fi}\ \{\ Q\ \}}\ \textsc{Cond1}$$

- $\{\ P \wedge B\ \}\ S\ \{\ Q\ \}$: $\{\ x = x_0 \wedge x < 0\ \}\ x := -x\ \{\ x \geq 0 \wedge (x = x_0 \vee x = -x_0)\ \}$ Backward construction (assignment axiom) gives the implication:

  $x = x_0 \wedge x < 0 \Rightarrow (-x \geq 0 \wedge (-x = x_0 \vee -x = -x_0))$

- $P \wedge \neg B \Rightarrow Q$: $x = x_0 \wedge x \geq 0 \Rightarrow (x \geq 0 \wedge (x = x_0 \vee x = -x_0))$