ID1217 Concurrent Programming
Lecture 3

Introduction to Axiomatic Semantics of Concurrent Programs with Shared Variables

ROYAL INSTITUTE OF TECHNOLOGY

Vladimir Vlassov
KTH/EECS

Properties of a Program

- ***A property*** of a program is an attribute of ALL histories of a program, e.g., correctness.
- Two kinds of properties: safety and liveness
- ***A safety property states that something bad never happens***
  - A safety property of a program is one, which must be *always* true, e.g., absence of deadlocks
  - The program *never* enters a *bad state*, in which the property does not hold, e.g., a deadlock state – the bad state is unreachable
- ***A liveness property states that something good eventually happens***
  - A liveness property is one, which the program *eventually* enters a *good state*, e.g., the program terminates

Properties (cont'd)

- A property, e.g. *correctness*, can be represented as a combination (conjunction) of a safety property and a liveness property
- For example:
  – Correctness: A message sent is *always* delivered *exactly once*
  – Liveness: A message sent is delivered *at least once*
    – Something *good* – delivered – *eventually* happens
  – Safety: A message sent is delivered *at most once*
    – Something *bad* – delivered twice – *never* happens

Examples of Safety Properties

Something bad never happens

- Partial correctness
  - The final state is correct, assuming that the program terminates
  - No incorrect results if the program terminates
- Mutual exclusion of critical sections
  - More than one thread should never execute a critical section at same time
  - Bad when more than one proc. are in critical sessions
- Absence of deadlocks (a.k.a. deadlock freedom)
  - A set of processes is deadlocked when each process in the set is waiting for an event which can only be caused by another process in that set.

Examples of Liveness Properties

Something good eventually happens

- Termination: the program *eventually* terminates
  - Every history (trace) is finite.
- Every competing process *eventually* enters a critical section

Total Correctness Property

- The ***total correctness property*** combines termination (liveness) and partial correctness (safety)
  - A program always terminates with a correct result

Proving Properties

Three main approaches

1. Testing or debugging
   - Run and see what happens
   - Limited to considered cases
2. Operational reasoning
   - "Exhaustive case analysis"
   - Considers enormous number of histories: $n \cdot m! / (m!)^n$
   - Helps in development
3. **Assertional reasoning**
   - Based on axiomatic semantics: axioms, inference rules, assertions
   - Work is proportional to the number of atomic actions

A Formal Logical System

- A set of ***symbols***
- A set of ***formulas*** constructed from symbols
- A set of ***axioms***, i.e. formulas which are (assumed to be) true
- A set of ***inference rules*** used to derive new true formulas (conclusions C) from other true formulas (hypotheses H)

$$\frac{\boldsymbol{H_1, H_2, \ldots, H_n}}{\boldsymbol{C}}$$

  – An inference rule is a procedure which combines known facts (Hs) to produce ("infer") new facts (C)
- ***Proof*** is a sequence of lines each of which either axiom or can be derived from previous by applying one of inference rules.
- ***Theorem*** is a line in a proof which is not an axiom
- Axioms have no premises

Interpretation of a Logic. Soundness and Completeness

- ***Interpretation*** maps each formula to true or false
  - Formulas: statements about some domain of discourse
  - Interpretation is a ***model*** for the logic if the logic is sound w.r.t. interpretation
- Logic is ***sound*** w.r.t. interpretation if all axioms and inference rules are sound
  - An axiom is sound if it maps to true
  - An inference rule is sound if C maps to true assuming Hs
- A logic is ***complete*** w.r.t. interpretation if every formula that maps to true is provable in the logic

Programming Logic (PL)

- What? Formal logical system that allows to state and to prove properties of programs.
- Why?
  - Allows proving safety properties (e.g. partial correctness) of concurrent programs.
  - Provides a systematic way to understand and to develop correct (reliable) concurrent programs

PL (cont'd)

- Formulas are ***triples*** of the form

  `{ P }  S  { Q }`

  - **`P`**, **`Q`** are predicates (assertions)
  - **`S`** is a sequence of program statements
- A predicate tells about a state (before/after execution)

  `{ x == 0 and y == 0 }`
- Interpretation defines relation between **`P`**, **`Q`** and **`S`**
  - **`P`** is called the ***precondition*** (state before execution)
  - **`Q`** is called the ***postcondition*** (state after execution)
  - Extreme assertions: true (all states), false (no state)

Interpretation of a Triple

- A triple **`{P} S {Q}`** is referred as partial correctness statement (or partial correctness assertion triple)
  - The triple **`{P} S {Q}`** is true if, whenever execution of **`S`** starts in a state satisfying **`P`** and execution of **`S`** terminates, the resulting state satisfies **`Q`**.
- Examples:

```
{ x == 0 }  x = x + 1; { x == 1 }
```

  - Should be a theorem (axiom)

```
{ x == 0 }  x = x + 1; { y == 1 }
```

  - Should not be a theorem (does not sound)

Recall: Implication Operation

- The implication operator (IMPLIES) is a binary operator that is typically denoted it with an arrow ("->“):

P	Q	P -> Q
False	False	**True**
False	True	**True**
True	False	**False**
True	True	**True**

- So `P -> Q` follows the following reasoning:
  - A True premise implies a True conclusion: T -> T is T;
  - A True premise cannot imply a False conclusion: T -> F is F;
  - One can conclude anything from a false assumption: F -> anything is T.

Assignment Axiom

$$\{ P_{x \leftarrow e} \} \;\; \mathtt{x = e} \;\; \{ P \}$$

- $P_{x \leftarrow e}$ specifies textual substitution: replace all free occurrences of the variable **x** in **P** by expression **e**.
  - If you want a final state to satisfy **P**, then the prior state must satisfy **P** with **x** textually replaced by **e**
  - The more common way to view assignments is by “going forward”: start with precondition and produce the postcondition
- For example:

```
{1 == 1} x = 1 {x == 1}
{x == 0} x = 1 {x == 1}
```

Inference Rules in PL

- Used to characterize the effects of prog. statements

Composition rule:
$$\frac{\{P\}\ \mathtt{S1}\ \{Q\},\quad \{Q\}\ \mathtt{S2}\ \{R\}}{\{P\}\ \mathtt{S1;\ S2}\ \{R\}}$$

If Statement rule:
$$\frac{\{P \wedge B\}\ \mathtt{S}\ \{Q\},\ (P \wedge \neg B) \Rightarrow Q}{\{P\}\ \mathtt{if\ (B)\ S}\ \{Q\}}$$

While Statement rule:
$$\frac{\{I \wedge B\}\ \mathtt{S}\ \{I\}}{\{I\}\ \mathtt{while\ (B)\ S}\ \{I \wedge \neg B\}}$$

Rule of Consequence:
$$\frac{P' \Rightarrow P,\ \{P\}\ \mathtt{S}\ \{Q\},\quad Q \Rightarrow Q'}{\{P'\}\ \mathtt{S}\ \{Q'\}}$$

- The rule of consequence allows to modify the predicates in triples, i.e. allows to strengthen preconditions and to weaken postconditions

Semantics of Synchronization and Concurrent Execution

ROYAL INSTITUTE OF TECHNOLOGY

Await statement rule:

$$\frac{\{P \land B\}\ S\ \{Q\}}{\{P\}\ \langle\ \texttt{await}\ (B)\ S\ \rangle\ \{Q\}}$$

Co statement rule:

$$\frac{\{P_i\}\ S_i\ \{Q_i\} \quad \textbf{are interference free}}{\{P_1 \land \ldots \land P_n\}\ \texttt{co}\ S_1\ ||\ \ldots\ ||\ S_n\ \texttt{oc}\ \{Q_1 \land \ldots \land Q_n\}}$$

- For conclusion to be true, proofs of hypotheses must not interfere each other.
- A proc ***interferes*** with another proc if the former executes an assignment that invalidates an assertion of the latter.
  - Arises because of shared variables.

Noninterference

- Define:
  - An ***assignment action*** is an assignment statement or an await statement that contains assignments.
  - A ***critical assertion*** is a precondition or postcondition that is not within an await statement.
- ***Noninterference***:
  - Let **`a`** be an assignment action in one process and let **`pre(a)`** be its precondition.
  - Let **`C`** be a critical assertion in another process.
  - Then **`a`** does not interfere with **`C`** if the following is a theorem in programming logic

    $\{ C \wedge pre(a) \}\ a;\ \{ C \}$

Ways to Avoid Interference

1. Disjoint variables
   – Avoid false dependences.
2. Weakened assertions
   – Say less than you could in isolation, take into account concurrency.
3. Global invariants
   – Predicates that are true in all visible states.
4. Synchronization
   – Hide states and/or delay execution.

1. Disjoint Variables

- Recall: P_1 and P_2 **do not depend** on each other if
  $$\mathbf{ws_1 \cap (rs_2 \cup ws_2) = \varnothing \text{ AND } ws_2 \cap (rs_1 \cup ws_1) = \varnothing}$$
- ***Reference set (refs)*** is formed of variables that appear in assertions in a proof
- Assertions should not capture false dependences!
- This implies: P_1 and P_2 **do not interfere** with each if
  $$\boldsymbol{ws_1 \cap refs_2 = \varnothing \text{ AND } ws_2 \cap refs_1 = \varnothing}$$

2. Weakened Assertions

- Take into account concurrency and say less than you could in isolation
  - A weakened assertion admits more program states than another assertion of a process in isolation
- For example:

```
{x == 0}
co {x == 0 ∨ x == 2} <x = x + 1;> {x == 1 ∨ x == 3}
|| {x == 0 ∨ x == 1} <x = x + 2;> {x == 2 ∨ x == 3}
oc
{x == 3}
```

  - The pre(post)condition of the program is the conjunction of the pre(post)conditions of the processes, e.g. the postcondition:

    `{x == 1 ∨ x == 3} ∧ {x == 2 ∨ x == 3} = {x == 3}`

3. Global Invariants

- Suppose **`I`** is a predicate that references global (shared) variables
- The predicate **`I`** is a ***global invariant*** for a set of processes if
  - **`I`** is true when the processes begin execution,
  - **`I`** is preserved by every assignment action.
- If every critical assertion **`C`** in the proof of every process P_i has the form **`I`** $\wedge$ **`L`**,
  - where **`L`** is a predicate on local vars in P_i, i.e. every var in **`L`** is assigned by only P_i,

then the proof of the processes P_i is interference-free.

4. Synchronization

- Used to delay processes by strengthening preconditions with additional constraints to avoid interference
  - A process waits until a stronger precondition is true, i.e. until its precondition taken in isolation is true and there is no interference with other processes
- Used to execute a set of statements as an atomic action.
  - To hide internal states.
  - To access shared variables with mutual exclusion.
- For example, consider the following:

```
co  …; a; …
|| …; {C} S2; …
oc
```

  - Here, the assignment **a** interferes with **C**. To avoid interference, use condition synchronization :

```
<await (!C or B) a;>
```

  - Here, **B** describes states such that executing **a** makes **C** true.

Approaches to Proving Safety Properties with PL

1. Avoid “bad” states
   - Assume ***BAD*** is a predicate that defines a “bad“ program state according to some property ***P***
     - The program deadlocks
     - More than one process enter critical section
   - The program satisfies ***P*** iff ***BAD*** is false in every history.
2. Be always in “good” states
   - Assume ***GOOD* = ¬*BAD*** is a predicate that defines a “good“ program state according to some property ***P***
   - The program satisfies ***P*** if ***GOOD*** is its global invariant.

Proving Safety Properties (cont'd)

3. Exclusion of configurations

```
co #process 1
   …; {pre(S1)} S1; …
|| # process 2
   …; {pre(S2)} S2; …
oc
```

- Where: `pre(S1)`$\wedge$ `pre(S2) == false` – the "bad" state
- Two processes cannot be at these statements (S1 and S2) at the same time.
- Helps to prove absence of deadlock

Example: Producer-Consumer

- Copy **`a[n]`** in **`Producer`** to **`b[n]`** in **`Consumer`** using a shared single-slot buffer **`buf`**
  - **`Producer`** writes **`a[p]`**, **`p`** = **`0`**, **`1`**, **`2`**, …, **`n`** to the **`buf`**
  - **`Consumer`** reads **`buf`** to **`b[c]`**, **`c`** = **`0`**, **`1`**, **`2`**, …, **`n`**
  - Condition synchronization to alternate access to the buffer
  - The synchronization requirement (predicate):

$$PC: \mathtt{c} \le \mathtt{p} \le \mathtt{c} + 1$$

where **`p`** – number of produced (stored)

`c` – number of consumed (fetched)

Consumer-Producer (cont' d)
A Parallel Program

```
int buf, p = 0, c = 0;
process Producer {
   int a[n];
   while (p < n) {
      < await (p == c) >
      buf = a[p];
      p = p+1;
   }
}
```

```
process Consumer {
   int b[n];
   while (c < n) {
      < await (p > c) >
      b[c] = buf;
      c = c+1;
   }
}
```

Consumer-Producer (cont’d)
Global Invariant of the Application

The synchronization requirement (predicate):

$$c \le p \le c + 1$$

Global invariant:

PC: (`c ≤ p ≤ c + 1`) ∧ (`a[0:n-1] == A[0:n-1]`) ∧
(`p == c+1`) ⇒ (`buf == A[p-1]`)

– where **`A[n]`** are values stored in **`a[n]`**

1) **`buf`** is either full (**`p == c+1`**) or empty (**`p == c`**)
2) **`a`** is not altered
3) When **`buf`** is full (**`p == c+1`**), it contains a value (**`A[p-1]`**)

Consumer-Producer (cont' d) Proof Outline

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1])}

process Producer {
  int a[n];    # assume a[i] is initialized to A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩   # delay until buffer empty
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
  {IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩    # delay until buffer full
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

- This ***proof outline*** captures the essence of what is true at each point
- It's an encoding of an actual proof in PL

PC: global invariant
IC: while loop invariant in Consumer
IP: while loop invariant in Producer

Consumer-Producer (cont'd): Proving a Safety Property

Let's prove absence of deadlock (when both are waiting – "bad" state)

- When Producer in its **await** statement waiting for **buf** to be empty, the following is true:

$$WP = PC \wedge (p < n) \wedge (p \neq c)$$

  – It's not finished yet and the buffer is full

- When Consumer in its **await** statement waiting for **buf** to be full, the following is true:

$$WC = IC \wedge (c < n) \wedge (p \leq c)$$

  – It's not finished yet and the buffer is empty

- Conclusion: Deadlock cannot occur because

$$WC \wedge WP = (p \neq c) \wedge (p == c) = false$$

  – We've used the approach of exclusion of configurations

Example: Noninterference

- Exercise 2.16 in the MPD book: Consider the following fragment:

```
int x = 0;
  co < await (x != 0) x = x - 2; >
  // < await (x != 0) x = x - 3; >
  // < await (x == 0) x = x + 5; > oc
```

  – Prove that the final value of x is 0.

  – Identify which assertions are critical, and show that they are not interfered with.

- Proof outline:

```
{true} int x = 0; {x == 0}
co
  {x == 0 or x == 5 or x == 2}    #1
  < await (x != 0) {x == 5 or x == 2} x = x - 2; {x == 3 or x == 0} >
  {x == 3 or x == 0}              #2
//
  {x == 0 or x == 5 or x == 3}    #3
  < await (x != 0) {x == 5 or x == 3} x = x - 3; {x == 2 or x == 0} >
  {x == 2 or x == 0}              #4
//
  {x == 0}                        #5
  < await (x == 0) {x == 0} x = x + 5; {x == 5} >
  {x == 0 or x == 5}              #6
oc
{x == 0}
```

  – Here, assertions 1, 2, 3, 4, 5, 6 are critical.

ROYAL INSTITUTE OF TECHNOLOGY

Example (cont'd) Proof of Noninterference

- The assertion 1 is not interfered (violated) by process 2.

  `{x == 0 or x == 5 or x == 2}` `and {x == 0 or x == 5 or x == 3}`

  `< await (x != 0) {x == 5} x = x - 3; {x == 2} >`

  `{x == 2} =>` **`{x == 0 or x == 5 or x == 2}`**

- The assertion 2 are not interfered (violated) by process 3. The theorem:

  `{x == 3 or x == 0}` `and` {1st await is executed first} `and {x == 0}`

  `< await (x == 0) {x == 0} x = x + 5; {x == 5} >`

  `{x == 3 or x == 0}`

  – Here, "1st await is executed first" can be expressed as

  `{x == 0}` `< await (x != 0) x = x - 2; >` **`{FALSE}`**

  that is FALSE

  – The entire triple is FALSE, therefore the 3rd await does not interfere with the postcondition of the 1st await, because the 1st await is executed after the 3rd one.

- Other theorems are proved in similar way. There are in total 12 theorems here.

Scheduling Policies and Fairness

- A ***scheduling policy*** determines which of actions eligible for execution will be executed next.
- Most liveness properties depend on fairness of the scheduling policy
- ***Fairness*** is a property of a scheduling policy which insures that a process (eventually) gets the chance to proceed regardless of what other processes do.

Fairness of Scheduling Policies

- A scheduling policy is ***unconditionally fair*** if every unconditional atomic action that is eligible is executed eventually
- A scheduling policy is ***weakly fair*** if
  1) it is unconditionally fair, and
  2) every conditional atomic action that is eligible is executed eventually, assuming that its condition becomes true and then remains true until it is seen by the process executing the conditional atomic action.
     - The program should guarantee that the above assumption is true

Fairness of Scheduling Policies (cont'd)

- A scheduling policy is ***strongly fair*** if
  1) it is unconditionally fair, and
  2) every conditional atomic action that is eligible is executed eventually, assuming that its condition is infinitely often true.

- Round-robin and time-slicing are practical but not strongly fair

ROYAL INSTITUTE OF TECHNOLOGY

Example

```
boolean cont = true, try = false;
co
   while (cont) {
     try = true;
     try = false;
   }
|| < await (try) cont = false; >
oc
```

- Does this program terminate?
  - With a weakly fair policy – might not terminate because **`try`** is also infinitely often **`false`**
  - With a strongly fair policy – will eventually terminates because **`try`** is infinitely often **`true`**