COMP6210 Automated Software Verification

Modelling Programs with Transition Systems

Pavel Naumov

Outline

Modelling with Transition Systems

Intended Learning Outcomes

By the end of these two lectures, you will understand

- how a (concurrent) program can be represented using a formalism called transition systems, in order to carry out verification tasks
- what such verification amounts to, once a representation for the program has been constructed

Additional reading (not compulsory): Chapter 2 of Clarke et al, available from module wiki

- gives more details than required

Transition Systems as Models of Systems

- modelling a (software) system amounts to identifying:
  - its internal *state* at any given time,
  - which state(s) each state can *transition* into.
- informally, transition systems are graphs with nodes representing system states and edges representing *atomic* state changes
  - we label states with their relevant properties
- e.g. simple traffic light model:

$$\begin{tikzcd} & s_1(red, amber) \arrow[dr] & \\ s_0(red) \arrow[ur] & & s_2(green) \arrow[dl] \\ & s_3(amber) \arrow[ul] \arrow[loop below] & \end{tikzcd}$$

- paths through the graph correspond to system behaviours

Transition Systems, Formally

We use a set *Prop* of *atomic propositions* to describe basic properties of states.

- e.g. $Prop = \{red, amber, green\}$

A transition system T over *Prop* is given by:

- a finite set S of states
- a subset $S_0 \subseteq S$ of initial states
- a transition relation $R \subseteq S \times S$ between states
- a valuation $V : S \to \mathcal{P}(Prop)$ giving, for each state $s \in S$, the atomic propositions which are true in that state: $V(s) \subseteq Prop$

Transition Systems - an Example

$Prop = \{red, amber, green\}$

$$\begin{tikzcd}
 & s_1(red, amber) \arrow[dr] & \\
\longrightarrow s_0(red) \arrow[ur] & & s_2(green) \arrow[dl] \\
 & s_3(amber) \arrow[ul] \arrow[loop below] &
\end{tikzcd}$$

- set of states: $S = \{ s_0, s_1, s_2, s_3 \}$
- set of initial states: $\{ s_0 \}$
- transition relation: $\{ (s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_3), (s_3, s_0) \}$
  - notation: $s_0 \longrightarrow s_1$, $s_1 \not\longrightarrow s_0$
- valuation V gives labelling of states with atomic propositions:

$$V(s_0) = \{red\} \qquad V(s_1) = \{red, amber\}$$
$$V(s_2) = \{green\} \qquad V(s_3) = \{amber\}$$

Transition Systems as Models of Programs

- states model program states
  - this includes values stored in all memory (stack/heap) plus the program counter
- transitions model atomic computation steps (arising from executing atomic program statements)
- atomic propositions describe basic properties of states (e.g. values of program variables)
- *maximal* paths (i.e. paths that cannot be extended any further) starting in an initial state correspond to possible program executions !

Extracting Models from Sequential Programs - Example

```
l0 : int x = 0, y = 0;
l1 :     while (true)
l2 :         atomic {x = (x+1) mod 3; y = x+1;}
```

$$
\begin{tikzcd}
 & & s_0\ (l_0) \arrow[d] \\
 & & s_1\ (l_1, x{=}0, y{=}0) \arrow[d] \\
 & & s_2\ (l_2, x{=}0, y{=}0) \arrow[d] \\
s_7\ (l_1, x{=}0, y{=}1) \arrow[r] & s_8\ (l_2, x{=}0, y{=}1) \arrow[r] & s_3\ (l_1, x{=}1, y{=}2) \arrow[d] \\
s_6\ (l_2, x{=}2, y{=}3) \arrow[u] & s_5\ (l_1, x{=}2, y{=}3) \arrow[l] & s_4\ (l_2, x{=}1, y{=}2) \arrow[l]
\end{tikzcd}
$$

Note:

- transitions match *atomic* program statements (*single* program steps)
- we can use atomic propositions to describe
  - variable values, e.g. x=2 (true in s_5, s_6), y=2 (true in s_3, s_4),
  - values of the program counter, e.g. $PC = l_2$ (true in s_2).

Extracting Models from Concurrent Programs - Example (part 1)

Assume the initial value of x is 0.

```
x = x+1;  x = x+1
```

```
x = x-1;  x = x-1
```

s_0 (x=0)

$\downarrow$

s_1 (x=1)

$\downarrow$

s_2 (x=2)

s_0 (x=0)

$\downarrow$

s_1 (x=-1)

$\downarrow$

s_2 (x=-2)

Extracting Models from Concurrent Programs - Example (part 1)

Assume the initial value of x is 0.

a: x = x+1; b: x = x+1 c:

d: x = x-1; e: x = x-1 f:

$$
\begin{array}{c}
s_0 \ (\mathrm{a}, x=0) \\
\downarrow \\
s_1 \ (\mathrm{b}, x=1) \\
\downarrow \\
s_2 \ (\mathrm{c}, x=2)
\end{array}
\qquad\qquad
\begin{array}{c}
s_0 \ (\mathrm{d}, x=0) \\
\downarrow \\
s_1 \ (\mathrm{e}, x=-1) \\
\downarrow \\
s_2 \ (\mathrm{f}, x=-2)
\end{array}
$$

Extracting Models from Concurrent Programs - Example (part 2)

x = 0;

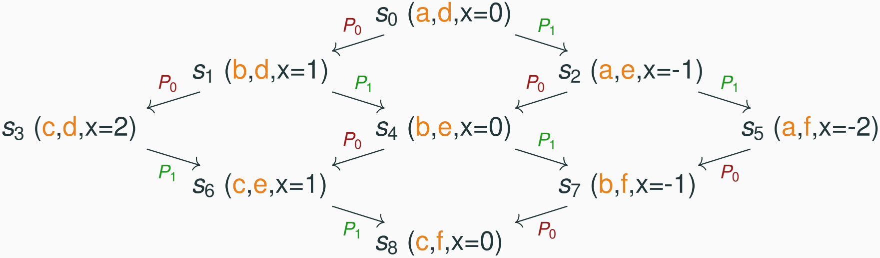
P_0: (a: x = x+1; b: x = x+1 c:) || P_1: (d: x = x-1; e: x = x-1 f:)

Extracting Models from Concurrent Programs - Example (part 2)

x = 0;

P_0: (a: x = x+1; b: x = x+1 c:) $\|$ P_1: (d: x = x-1; e: x = x-1 f:)

The following models the interleaved execution of the two processes:

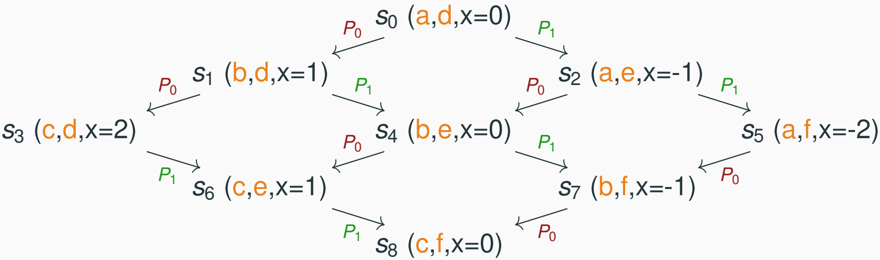


Extracting Models from Concurrent Programs - Example (part 2)

x = 0;

P_0: (a: x = x+1; b: x = x+1 c:) || P_1: (d: x = x-1; e: x = x-1 f:)

The following models the interleaved execution of the two processes:



The above transition system is nondeterministic: there are several ways of proceeding from a given state.

The Behaviour of a Transition System

Possible behaviours arise as follows:

- nondeterministically select an initial state s
- while s has outgoing transitions:
  - nondeterministically select a transition $s \to s'$
  - execute the corresponding action
  - let $s := s'$

System executions are thus *maximal* sequences:

$$s_0 \to s_1 \to s_2 \to \dots$$

(can be finite or infinite!)

Modelling Concurrent Programs

We will look at concurrent programs:

- several processes/threads executing concurrently and communicating through shared variables
- usual sequential constructs: assignments, **if**, **while**, **skip**, ...
- concurrency primitives: e.g. **wait**, **lock**, **unlock** statements

For example:

bool *turn*;

$P \;=\; \textbf{cobegin}\; P_0 \parallel P_1 \;\textbf{coend}$

$P_0 \;=\; \textbf{while}\,(\mathit{True})\{$
  `local_actions;` $\textbf{wait}\,(\mathit{turn} == 0);$
  `use_resource;` $\mathit{turn} = 1$
  $\}$

$P_1 \;=\; \textbf{while}\,(\mathit{True})\{$
  `local_actions;` $\textbf{wait}\,(\mathit{turn} == 1);$
  `use_resource;` $\mathit{turn} = 0$
  $\}$

- **cobegin** ... **coend** specifies concurrent execution
- **wait** (c) repeatedly tests condition c until it becomes true.

Why Model Checking?

The following program implements a simple mutual exclusion protocol.

$$
\begin{aligned}
&\textbf{bool}\ \mathit{turn};\\
&P = \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend}\\
&P_0 = \textbf{while}\,(\mathit{True})\{ \qquad\qquad\qquad\qquad P_1 = \textbf{while}\,(\mathit{True})\{\\
&\quad \texttt{local\_actions;}\ \textbf{wait}\,(\mathit{turn} == 0); \quad \texttt{local\_actions;}\ \textbf{wait}\,(\mathit{turn} == 1);\\
&\quad \texttt{use\_resource;}\ \mathit{turn} = 1 \qquad\qquad\quad \texttt{use\_resource;}\ \mathit{turn} = 0\\
&\quad \} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}
$$

We can use a model of this program to check:

- Can the program reach a state where both P_0 and P_1 are using the shared resource? (This would violate mutual exclusion.)
- Does there exist an execution of the program where P_1 never accesses the shared resource?

Note: shaded code does not influence above properties as long as it terminates and does not modify *turn*!

Adding Program Counters

Question: How do we represent such programs using transition systems?

Adding Program Counters

Question: How do we represent such programs using transition systems?

First step is to identify the (unique!) *entry* and *exit* points of each *atomic* program statement, e.g.

$$
\begin{array}{ll}
P_0 = n_0 : \textbf{while}\,(True)\{ & \\
\quad t_0 : \textbf{wait}\,(turn == 0); & \\
\quad c_0 : turn = 1 & \\
\}\; n_0' : &
\end{array}
\qquad
\begin{array}{ll}
P_1 = n_1 : \textbf{while}\,(True)\{ & \\
\quad t_1 : \textbf{wait}\,(turn == 1); & \\
\quad c_1 : turn = 0 & \\
\}\; n_1' : &
\end{array}
$$

Adding Program Counters

Question: How do we represent such programs using transition systems?

First step is to identify the (unique!) *entry* and *exit* points of each *atomic* program statement, e.g.

$$\begin{aligned} P_0 = n_0 :\ & \textbf{while}\,(True)\{ \\ & t_0 : \textbf{wait}\,(turn == 0); \\ & c_0 : turn = 1 \\ \}\ & n_0' : \end{aligned} \qquad \begin{aligned} P_1 = n_1 :\ & \textbf{while}\,(True)\{ \\ & t_1 : \textbf{wait}\,(turn == 1); \\ & c_1 : turn = 0 \\ \}\ & n_1' : \end{aligned}$$

For sequential programs, we define a recursive procedure for annotating a program with entry points of atomic program statements.

- why are entry points sufficient?
- **Note: if** and **while** statements are not atomic !

The annotation procedure for sequential programs

1. if P is not a composite statement (e.g. P is an assignment, **skip**, **wait**, **lock**, **unlock**), do nothing: $P^{\mathcal{L}} = P$
2. if $P = P_1 ; P_2$

   $$P^{\mathcal{L}} = P_1^{\mathcal{L}} ; l : P_2^{\mathcal{L}}$$

3. if $P = \textbf{if}(b)\ P_1\ \textbf{else}\ P_2$

   $$P^{\mathcal{L}} = \textbf{if}(b)\ l_1 : P_1^{\mathcal{L}}\ \textbf{else}\ l_2 : P_2^{\mathcal{L}}$$

4. if $P = \textbf{while}(b)\{\ P_1\ \}$

   $$P^{\mathcal{L}} = \textbf{while}(b)\{\ l_1 : P_1^{\mathcal{L}}\ \}$$

At the end, add labels for the entry and exit points of the program itself.

Example: Basic Mutual Exclusion Protocol

$$
\begin{aligned}
&n_0 : \ \textbf{while}\,(\mathit{True})\{\\
&\qquad t_0 : \ \textbf{wait}\,(\mathit{turn} == 0);\\
&\qquad c_0 : \mathit{turn} = 1\\
&\}\ n_0'
\end{aligned}
$$

The annotation procedure for concurrent programs

1. if $P = \textbf{cobegin } P_1 \parallel P_2 \parallel \ldots \parallel P_n \textbf{ coend}$

$$P^{\mathcal{L}} = \textbf{cobegin } l_1 : P_1^{\mathcal{L}} l_1' \parallel l_2 : P_2^{\mathcal{L}} l_2' \parallel \ldots \parallel l_n : P_n^{\mathcal{L}} l_n' \textbf{ coend}$$

Note:

- exit points of concurrent processes also need to be labelled - why?
- no two labels must be identical

Example: Basic Mutual Exclusion Protocol

$$
\begin{array}{l}
\textbf{bool}\ turn; \\
P \ = \ \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend} \\
P_0 \ = \ n_0 : \ \textbf{while}\,(True)\{ \\
\quad t_0 : \ \textbf{wait}\,(turn == 0); \\
\quad c_0 : \ turn = 1 \\
\}\ n_0' :
\end{array}
\qquad
\begin{array}{l}
P_1 \ = \ n_1 : \ \textbf{while}\,(True)\{ \\
\quad t_1 : \ \textbf{wait}\,(turn == 1); \\
\quad c_1 : \ turn = 0 \\
\}\ n_1' :
\end{array}
$$

Extract a transition system which models P:

- states are determined by the program counters of P_0 and P_1, together with the value of the shared variable *turn*
- transitions correspond to atomic execution steps in *one* of the processes (execution of processes is interleaved!)
- *Prop* and *V* are extracted from states (more on this later)

Example: Basic Mutual Exclusion Protocol

$$\textbf{bool}\ \mathit{turn};$$

$$P \;=\; \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend}$$

$$\begin{aligned} P_0 \;=\; & n_0 : \textbf{while}\,(\mathit{True})\{ \\ & \quad t_0 : \textbf{wait}\,(\mathit{turn} == 0); \\ & \quad c_0 : \mathit{turn} = 1 \\ & \}\ n_0' : \end{aligned}$$

$$\begin{aligned} P_1 \;=\; & n_1 : \textbf{while}\,(\mathit{True})\{ \\ & \quad t_1 : \textbf{wait}\,(\mathit{turn} == 1); \\ & \quad c_1 : \mathit{turn} = 0 \\ & \}\ n_1' : \end{aligned}$$

Extract a transition system which models P:

- states are determined by the program counters of P_0 and P_1, together with the value of the shared variable *turn*
- transitions correspond to atomic execution steps in *one* of the processes (execution of processes is interleaved!)
- *Prop* and V are extracted from states (more on this later)

Example: Basic Mutual Exclusion Protocol

$$\textbf{bool}\ \mathit{turn};$$
$$P \;=\; \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend}$$

$$\begin{array}{l} P_0 \;=\; n_0 : \textbf{while}\,(\mathit{True})\{ \\ \quad t_0 : \textbf{wait}\,(\mathit{turn} == 0); \\ \quad c_0 : \mathit{turn} = 1 \\ \}\ n_0' : \end{array}$$

$$\begin{array}{l} P_1 \;=\; n_1 : \textbf{while}\,(\mathit{True})\{ \\ \quad t_1 : \textbf{wait}\,(\mathit{turn} == 1); \\ \quad c_1 : \mathit{turn} = 0 \\ \}\ n_1' : \end{array}$$

Extract a transition system which models P:

- states are determined by the program counters of P_0 and P_1, together with the value of the shared variable *turn*
- transitions correspond to atomic execution steps in *one* of the processes (execution of processes is interleaved!)
- *Prop* and *V* are extracted from states (more on this later)

Example: Basic Mutual Exclusion Protocol

$$
\begin{aligned}
&\textbf{bool}\ \mathit{turn};\\
&P \;=\; \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend}
\end{aligned}
$$

$$
\begin{aligned}
P_0 \;=\; & n_0 : \textbf{while}\,(\mathit{True})\{\\
& \quad t_0 : \textbf{wait}\,(\mathit{turn} == 0);\\
& \quad c_0 : \mathit{turn} = 1\\
& \}\ n_0' :
\end{aligned}
\qquad
\begin{aligned}
P_1 \;=\; & n_1 : \textbf{while}\,(\mathit{True})\{\\
& \quad t_1 : \textbf{wait}\,(\mathit{turn} == 1);\\
& \quad c_1 : \mathit{turn} = 0\\
& \}\ n_1' :
\end{aligned}
$$

Extract a transition system which models P:

- states are determined by the program counters of P_0 and P_1, together with the value of the shared variable *turn*
- transitions correspond to atomic execution steps in *one* of the processes (execution of processes is interleaved!)
- *Prop* and V are extracted from states (more on this later)

Mutual Exclusion: the Model

States are of the form $(turn,\ PC_0,\ PC_1)$

$$
\begin{array}{l}
\textbf{bool}\ turn; \\
P \ = \ \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend} \\
P_0 \ = \ n_0 : \ \textbf{while}\,(True)\{ \\
\quad t_0 : \ \textbf{wait}\,(turn == 0); \\
\quad c_0 : turn = 1 \\
\}\ n_0' :
\end{array}
\qquad
\begin{array}{l}
P_1 \ = \ n_1 : \ \textbf{while}\,(True)\{ \\
\quad t_1 : \ \textbf{wait}\,(turn == 1); \\
\quad c_1 : turn = 0 \\
\}\ n_1' :
\end{array}
$$

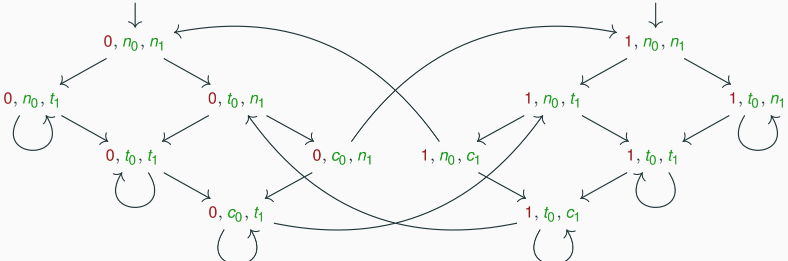
Mutual Exclusion: the Model

States are of the form ($turn$, PC_0, PC_1)

$$\textbf{bool}\ turn;$$
$$P = \textbf{cobegin}\ P_0 \parallel P_1\ \textbf{coend}$$

$$\begin{aligned} P_0 = n_0 :\ & \textbf{while}\,(True)\{ \\ & t_0 :\ \textbf{wait}\,(turn == 0); \\ & c_0 :\ turn = 1 \\ \}\ & n_0' : \end{aligned}$$

$$\begin{aligned} P_1 = n_1 :\ & \textbf{while}\,(True)\{ \\ & t_1 :\ \textbf{wait}\,(turn == 1); \\ & c_1 :\ turn = 0 \\ \}\ & n_1' : \end{aligned}$$



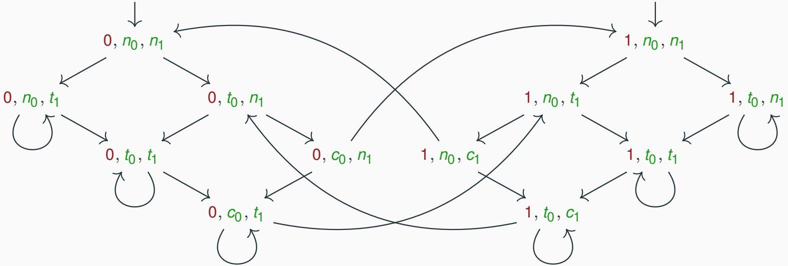
Extracting Transition Systems from Concurrent Programs

Outline of general procedure:

1. annotate the entry and exit points of basic program statements with process counters
2. the states of the transition system are tuples consisting of:
   - the values of global variables
   - the values of local process variables
   - the values of process counters
3. transitions between states correspond to individual atomic steps in one of the processes.
4. atomic propositions are of the following forms
   - $var = v$ with var a program variable and v a possible value for var
   - $PC_i = l$ with i a process and l the entry point of a statement in process i
     - so n_0 in the diagram is a shorthand for $PC_0 = n_0$

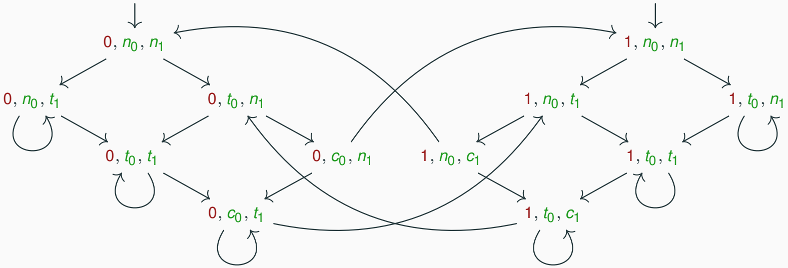
(See Chapter 2 of Clarke et al. for more details)

Back to Correctness Properties



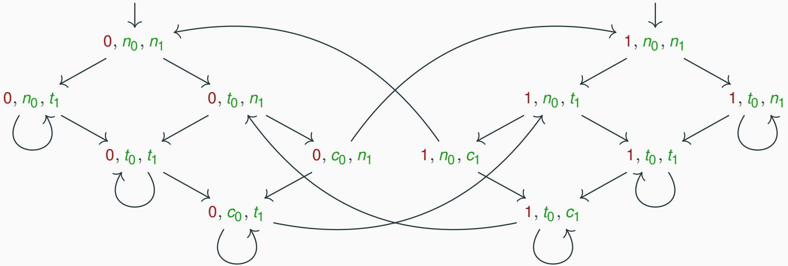
- Can the program reach a state where both P_0 and P_1 access the shared resource?

Back to Correctness Properties



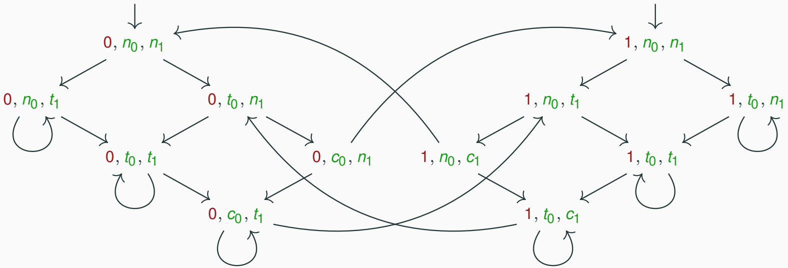
- Can the program reach a state where both P_0 and P_1 access the shared resource?
  - sufficient to check for a state s with $V(s) \ni PC_0 = c_0,\ PC_1 = c_1$

Back to Correctness Properties



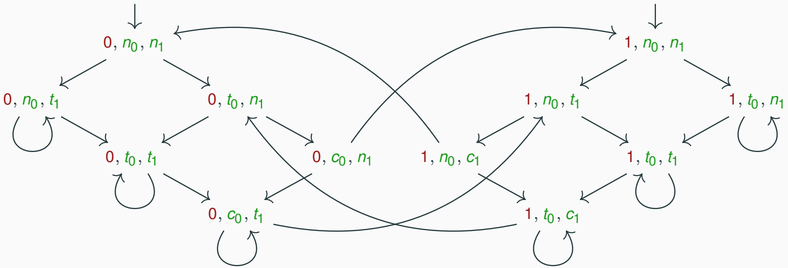
- Can the program reach a state where both P_0 and P_1 access the shared resource?
    - sufficient to check for a state s with $V(s) \ni PC_0 = c_0,\ PC_1 = c_1$
- Does there exist an execution of the program where P_1 never accesses the shared resource?

Back to Correctness Properties



- Can the program reach a state where both P_0 and P_1 access the shared resource?
  - sufficient to check for a state s with $V(s) \ni PC_0 = c_0,\ PC_1 = c_1$
- Does there exist an execution of the program where P_1 never accesses the shared resource?
  - sufficient to check if a path through the graph exists which starts in an initial state and never reaches states s with $V(s) \ni PC_1 = c_1$

Back to Correctness Properties



- Can the program reach a state where both P_0 and P_1 access the shared resource?
  - sufficient to check for a state s with $V(s) \ni PC_0 = c_0,\ PC_1 = c_1$
- Does there exist an execution of the program where P_1 never accesses the shared resource?
  - sufficient to check if a path through the graph exists which starts in an initial state and never reaches states s with $V(s) \ni PC_1 = c_1$
- We can verify program properties by exploring the above graph !

Some Important Concepts

- interleaving (of actions)
  - here used to model execution of concurrent processes sharing a single processor
  - but can also be used to model processes running on different machines
    - in this case the effect of executing two actions of different processes at the same time is the same as the effect of executing them one after the other. Why?
- nondeterminism (in the resulting transition system):
  - here caused by several possible interleavings
  - but can also be used to model program input, abstraction, . . .

Some Important Concepts

- interleaving (of actions)
  - here used to model execution of concurrent processes sharing a single processor
  - but can also be used to model processes running on different machines
    - in this case the effect of executing two actions of different processes at the same time is the same as the effect of executing them one after the other. Why?
- nondeterminism (in the resulting transition system):
  - here caused by several possible interleavings
  - but can also be used to model program input, abstraction, ...

Some Questions

- The previous model assumes that both processes spend only a finite amount of time executing local actions, or while in the critical section.

  How would you model the possibility that a process does not relinquish the shared resource after a finite amount of time?

- Is it *fair* that, in some infinite program executions, P_0 has infinitely many opportunities to execute, but never does? Or it only executes for a finite number of times?

  Fairness assumptions are used in model checkers to ignore such executions.

Some Questions

- The previous model assumes that both processes spend only a finite amount of time executing local actions, or while in the critical section.

  How would you model the possibility that a process does not relinquish the shared resource after a finite amount of time?

- Is it *fair* that, in some infinite program executions, P_0 has infinitely many opportunities to execute, but never does? Or it only executes for a finite number of times?

  Fairness assumptions are used in model checkers to ignore such executions.

Some Questions

- The previous model assumes that both processes spend only a finite amount of time executing local actions, or while in the critical section.

  How would you model the possibility that a process does not relinquish the shared resource after a finite amount of time?

- Is it *fair* that, in some infinite program executions, P_0 has infinitely many opportunities to execute, but never does? Or it only executes for a finite number of times?

  Fairness assumptions are used in model checkers to ignore such executions.

Fairness Assumptions

- capture the idea that the scheduling of processes is fair
    - if a process *can* run, it will eventually run
- this can be *assumed* when running a verification !
    - with fairness assumptions, only *fair* executions are explored by the model checker
- often needed to prove liveness properties
    - unfair executions often do not satisfy liveness properties

Fairness Assumptions

- capture the idea that the scheduling of processes is fair
  - if a process *can* run, it will eventually run
- this can be *assumed* when running a verification !
  - with fairness assumptions, only *fair* executions are explored by the model checker
- often needed to prove liveness properties
  - unfair executions often do not satisfy liveness properties

Correctness Properties

Assumption: programs may (and some are intended to) run forever.

Types of correctness properties:

- safety properties: nothing "bad" ever happens (in any possible program execution)
  - e.g. absence of deadlock: a program *never* enters a state it cannot leave
  - e.g. system invariants: some (desirable) property is true *in all reachable program states*
- liveness properties: something "good" eventually happens (in every possible execution of the program)
  - e.g. some property holds *eventually* / *infinitely often*
  - e.g. responsiveness: each request is eventually followed by a reply

Correctness Properties

Assumption: programs may (and some are intended to) run forever.

Types of correctness properties:

- safety properties: nothing "bad" ever happens (in any possible program execution)
  - e.g. absence of deadlock: a program *never* enters a state it cannot leave
  - e.g. system invariants: some (desirable) property is true *in all reachable program states*
- liveness properties: something "good" eventually happens (in every possible execution of the program)
  - e.g. some property holds *eventually* / *infinitely often*
  - e.g. responsiveness: each request is eventually followed by a reply

Safety versus Liveness

- if an execution violates a safety property, then it has a *finite prefix* which also violates the property

Safety properties are violated in finite time.

- above is not true of liveness properties!

Liveness properties are violated in infinite time.

Safety versus Liveness

- if an execution violates a safety property, then it has a *finite prefix* which also violates the property

Safety properties are violated in finite time.

- above is not true of liveness properties!

Liveness properties are violated in infinite time.

Exercise

Construct the transition system for the following program:

int x;

P = $x = 0$; **cobegin** Inc || Dec || Reset **coend**

Inc = **loop_forever** { **wait** $(x < 2)$; $x = x + 1$ }

Dec = **loop_forever** { **wait** $(x > 0)$; $x = x - 1$ }

Reset = **loop_forever** { **wait** $(x == 2)$; $x = 0$ }

Note:

- **loop_forever** similar to **while** statement but no condition to test
- **cobegin** ... **coend** statement now part of another program!
  - when reaching this statement, a transition is made from its entry point to the entry points of the individual processes
  - once *all* concurrent processes have finished executing (in the above case, never!), a transition is made to the exit point of the **cobegin** ... **coend** statement

Exercise

Construct the transition system that corresponds to the following program:

bool x, b_0, b_1;

$P = b_0, b_1 = 0, 0;$
cobegin $P_0 \parallel P_1$ **coend**

$P_0 =$ **loop_forever** $\{$
$b_0, x = 1, 1;$
wait $((b_1 == 0) \parallel (x == 0));$
$b_0 = 0$
$\}$

$P_1 =$ **loop_forever** $\{$
$b_1, x = 1, 0;$
wait $((b_0 == 0) \parallel (x == 1));$
$b_1 = 0$
$\}$

Note: "$b_0, b_1 = 0, 0$" is a concurrent assignment (atomic!)

Mutual Exclusion Example: the Specification

Correctness properties:

- **mutual exclusion** (safety): at most one process in critical section *at any time*
- **starvation freedom** (liveness): *whenever* a process tries to enter its critical section, it will *eventually* succeed
- **no strict sequencing**: processes need not enter their critical section in strict sequence (i.e. $P_0\ P_1\ P_0\ P_1 \ldots$ or $P_1\ P_0\ P_1\ P_0 \ldots$)

Exercise

Consider the previous (models of) mutual exclusion protocols.

Check (by inspecting the models) whether the three correctness properties hold in these models.

Does this depend on any fairness assumptions?

Historical and Bibliographical Notes

- Unfolding programs into transition systems has been used for verification of programs since the 1970s, e.g. *Formal verification of Parallel Programs*, R. M. Keller, in Comm. ACM, 19(7), 1976.
- The transitions systems covered in this lecture are also called Kripke structures after the American philosopher and logician Saul Kripke.
- This lecture covers Chapter 2, Modeling Systems, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.
- You may read more on this topic from Chapter 2, Modelling Concurrent Systems, from *Principles of Model Checking*, C. Baier, J.-P. Katoen.