Model Checking

Transition Systems

[Baier & Katoen, Chapter 2]

Joost-Pieter Katoen and Tim Quatmann

Software Modeling and Verification Group

RWTH Aachen, SoSe 2022

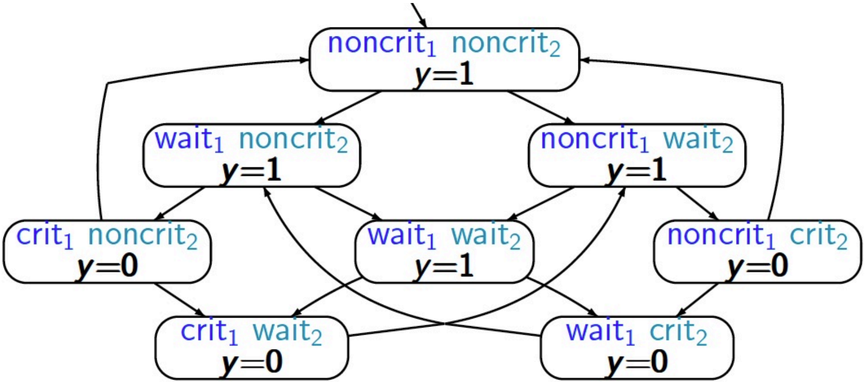
Overview

Overview

Transition systems

- Model to describe the behaviour of systems
- Digraphs where nodes represent states, and edges model transitions
- State:
  - the current colour of a traffic light
  - the current values of all program variables + the program counter
  - the current value of the registers plus the values of the input bits
- Transition: ("state change")
  - a switch from one colour to another
  - the execution of a program statement
  - the change of the registers and output bits for a new input

A Mutual Exclusion Algorithm



For simplicity, actions are omitted in this example.

Transition system

Definition: Transition system

A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states
- Act is a set of actions
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labelling function

S and Act are either finite or countably infinite

Notation: $s \xrightarrow{\alpha} s'$ as abbreviation of $(s, \alpha, s') \in \longrightarrow$

Direct Successors and Predecessors

$$Post(s, \alpha) = \left\{ s' \in S \;\middle|\; s \xrightarrow{\alpha} s' \right\}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

$$Pre(s, \alpha) = \left\{ s' \in S \;\middle|\; s' \xrightarrow{\alpha} s \right\}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha).$$

$$Post(C, \alpha) = \bigcup_{s \in C} Post(s, \alpha), \quad Post(C) = \bigcup_{s \in C} Post(s) \text{ for } C \subseteq S.$$

$$Pre(C, \alpha) = \bigcup_{s \in C} Pre(s, \alpha), \quad Pre(C) = \bigcup_{s \in C} Pre(s) \text{ for } C \subseteq S.$$

State s is called terminal if and only if $Post(s) = \varnothing$

Transition System "Behaviour"

The possible behaviours of a TS result from:

select non-deterministically an initial state $s \in I$

`while` s is not a terminal

`do`

  select non-deterministically a transition $s \xrightarrow{\alpha} s'$

  perform the action α and set $s = s'$

`od`

Executions

Definition: Executions

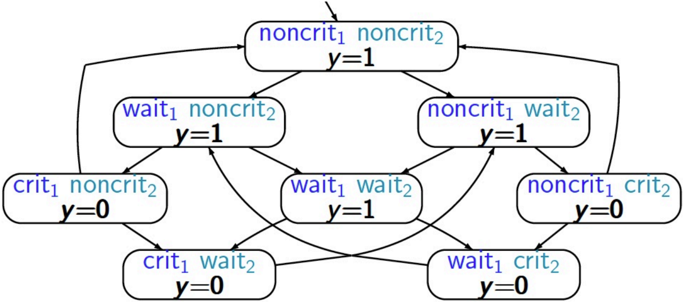
- An execution fragment $\rho \in (S \times Act)^{\omega}$ of transition systems TS is an infinite, alternating sequence of states and actions:

$$\rho \ = \ s_0 \, \alpha_1 \, s_1 \, \alpha_2 \, s_2 \, \alpha_3 \ldots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i.$$

  We also denote ρ by: $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$.
- ρ is maximal if ρ is infinite or finite and ending in a terminal state.
- ρ is initial if it starts in an initial state, i.e., $s_0 \in I$.
- An execution is an initial, maximal execution fragment.
- Omitting the actions from an execution yields a path.
- $Paths(s)$ is the set of all paths $\pi = s_0 s_1 s_2 \ldots$ starting in $s_0 = s \in S$.

A state s is reachable in TS if s occurs in some execution of TS.

Example Executions



Transition Systems versus Finite Automata

As opposed to finite automata, a transition system:

- has no accept/final states
- is not "accepting" a (regular) language
- may have countably infinite set of states and actions
- may be infinitely branching
- actions are used to "glue" small transition systems

Transition systems are used to model reactive systems, i.e., systems that continuously interact with their environment.

Overview

1. What are Transition Systems?
2. Traces
3. Program Graphs
4. Multi-Threading
5. Other Forms of Concurrency
6. The State Explosion Problem

Traces

- Actions are mainly used to model the (possibility of) interaction
  synchronous or asynchronous communication
- Here, focus on the states that are visited during executions
  the states themselves are not "observable", but just their atomic propositions
- Traces are sequences of the form $L(s_0)\, L(s_1)\, L(s_2) \ldots$
  record the (sets of) atomic propositions along an execution
- For transition systems without terminal states[1]:
  traces are infinite words over the alphabet 2^{AP}, i.e., they are in $\left(2^{AP}\right)^{\omega}$

[1] This is an assumption commonly used throughout this lecture.

Traces

Definition: Traces

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be transition system without terminal states.

- The trace of execution

$$\rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

  is the infinite word $trace(\rho) \;=\; L(s_0)\, L(s_1)\, L(s_2) \ldots$ over $\left(2^{AP}\right)^{\omega}$.
  Prefixes of traces are finite traces.

- The traces of a set Π of executions (or paths) is defined by:

$$trace(\Pi) \;=\; \{\, trace(\pi) \mid \pi \in \Pi \,\}.$$

- The traces of state s are $Traces(s) = trace(Paths(s))$.
- The traces of transition system TS: $Traces(TS) = \bigcup_{s \in I} Traces(s)$.

Example Traces

Consider the mutex transition system. Let $AP = \{ crit_1, crit_2 \}$.
The trace of the path:

$$\begin{aligned}\pi \quad = \quad & \langle n_1, n_2, y = 1 \rangle \rightarrow \langle w_1, n_2, y = 1 \rangle \rightarrow \langle c_1, n_2, y = 0 \rangle \rightarrow \\ & \langle n_1, n_2, y = 1 \rangle \rightarrow \langle n_1, w_2, y = 1 \rangle \rightarrow \langle n_1, c_2, y = 0 \rangle \rightarrow \ldots\end{aligned}$$

is:

$$trace(\pi) \;=\; \varnothing\, \varnothing \{ crit_1 \} \varnothing\, \varnothing \{ crit_2 \} \varnothing\, \varnothing \{ crit_1 \} \varnothing\, \varnothing \{ crit_2 \} \ldots$$

The finite trace of the finite path fragment:

$$\begin{aligned}\widehat{\pi} \quad = \quad & \langle n_1, n_2, y = 1 \rangle \rightarrow \langle w_1, n_2, y = 1 \rangle \rightarrow \langle w_1, w_2, y = 1 \rangle \rightarrow \\ & \langle w_1, c_2, y = 0 \rangle \rightarrow \langle w_1, n_2, y = 1 \rangle \rightarrow \langle c_1, n_2, y = 0 \rangle\end{aligned}$$

is:

$$trace(\widehat{\pi}) \;=\; \varnothing\, \varnothing\, \varnothing \{ crit_2 \} \varnothing \{ crit_1 \}$$

Overview

Transition Systems are Universal

Transition systems can model the behaviour of:

- Sequential programs
- Multi-threaded programs
- Communicating sequential programs
- Sequential hardware circuits
- Petri nets
- State Charts
- . . . and many more

Program Graphs

Let *Var* be a collection of typed variables over domain $\mathbb{D}$.

A program graph is a finite, rooted directed graph with:

- a finite set *Loc* of vertices, called locations
- a set of initial vertices (roots), called initial locations
- a set of labelled edges that connect locations with:
  - a Boolean condition over variables, e.g., $x < 10$
  - an action $\alpha \in Act$, e.g., $x := x{+}1$

  Intuition: `if` $x < 10$ `then` $x := x{+}1$
- an effect function describing the effect of an action on a variable valuation $\eta : Var \to \mathbb{D}$, e.g.,

$$Effect\big(x := x{+}1, \underbrace{[x \mapsto 5, y \mapsto 0]}_{\eta(x)=5, \eta(y)=0}\big) = \underbrace{[x \mapsto 6, y \mapsto 0]}_{\eta'(x)=6, \eta'(y)=0}$$

- an initial Boolean condition, e.g., $x{=}10 \wedge y < 3$

Program Graphs

Definition: Program graph

A program graph PG over set Var of typed variables is a tuple

$$(Loc, Act, Effect, \longrightarrow, Loc_0, g_0) \quad \text{where}$$

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function
- $\longrightarrow \subseteq Loc \times \underbrace{Cond(Var)}_{\text{Boolean conditions over } Var} \times Act \times Loc$ is the edge relation
- $g_0 \in Cond(Var)$ is the initial condition.

Notation: $\ell \xrightarrow{g:\alpha} \ell'$ denotes $(\ell, g, \alpha, \ell') \in \longrightarrow$

Example

Pseudo-code thread $i \in \{0, 1\}$:

```
     int k := 0;
     b := [true, true];
ℓ0 { while (true) do
       b[i] := false;
ℓ1 {   while (k != i) do
ℓ2 {     while (not b[1-i]) do
           k := i;
         end
       end
ℓ3 {   critical_section;
ℓ4 {   b[i] := true;
     end
```

initially b = [true, true]

and k = 0

Program Graphs $\mapsto$ Transition Systems

- Basic strategy: unfolding
  - state $=$ location (current control) ℓ $+$ valuation η
  - initial state $=$ initial location satisfying the initial condition
- Propositions and labelling
  - propositions: "at ℓ" and "$x \in D$" for $D \subseteq dom(x)$
  - $\langle \ell, \eta \rangle$ is labelled with "at ℓ" and all conditions that hold in η
- If $\ell \xrightarrow{g:\alpha} \ell'$ and g holds for the current valuation η, then

$$\underbrace{\langle \ell, \eta \rangle}_{\text{current state}} \quad \xrightarrow{\alpha} \quad \underbrace{\langle \ell', \mathit{Effect}(\alpha, \eta) \rangle}_{\text{next state}}$$

Program Graphs $\mapsto$ Transition Systems

Definition: Transition system of a program graph

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \longrightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \longrightarrow, I, AP, L)$ where

- $S = Loc \times Eval(Var)$
- $\longrightarrow \subseteq S \times Act \times S$ is defined by the rule:

$$\frac{\ell \xrightarrow{g:\alpha} \ell' \quad \wedge \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0\}$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

Example: Dijkstra's Mutual Exclusion

Pseudo-code thread *i*:

int k := 0;
b := [true, true];
ℓ_0 while (true) do
    $b[i] := false$;
ℓ_1 while (k != i) do
ℓ_2 while (not b[1-i]) do
        k := i;
      end
    end
ℓ_3 critical_section;
ℓ_4 b[i] := true;
end

initially b = [true, true]

and k = 0

Overview

Modelling Multi-Threading

- Transition systems
  - suited for modelling sequential programs
  - and for modelling sequential hardware circuits
- How about concurrent systems?
  - multi-threading
  - distributed algorithms and communication protocols
- Can we model:
  - multi-threaded programs with shared variables?

Interleaving

- Abstract from decomposition of system in threads

- Actions of independent threads are merged or "interleaved"
  - a single processor is available
  - on which the actions of the threads are interlocked

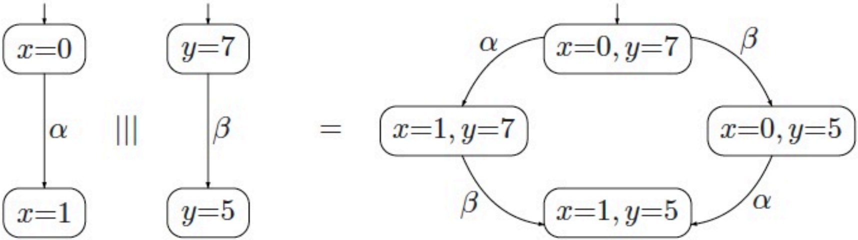
- No assumptions are made on the order of threads
  - possible orders for non-terminating independent threads P and Q:

$$
\begin{array}{cccccccccc}
P & Q & P & Q & P & Q & Q & Q & P & \ldots \\
P & P & Q & P & P & Q & P & P & Q & \ldots \\
P & Q & P & P & Q & P & P & P & Q & \ldots
\end{array}
$$

  - assumption: there is a scheduler with an a priori unknown strategy

Justification

$$\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\beta}$$



the effect of concurrently executed, independent actions α and β is equal regardless of their execution order

Interleaving of transition systems

Definition: Interleaving of transition systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ $i{=}1,2$, be two transition systems.

Transition system

$$TS_1 \,|||\, TS_2 = (S_1 \times S_2, Act_1 \uplus Act_2, \rightarrow, I_1 \times I_2, AP_1 \uplus AP_2, L)$$

where $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$ and the transition relation $\rightarrow$ is defined by the inference rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

Interleaving of Program Graphs

For program graphs PG_1 (on Var_1) and PG_2 (on Var_2) without shared variables, i.e., $Var_1 \cap Var_2 = \varnothing$,

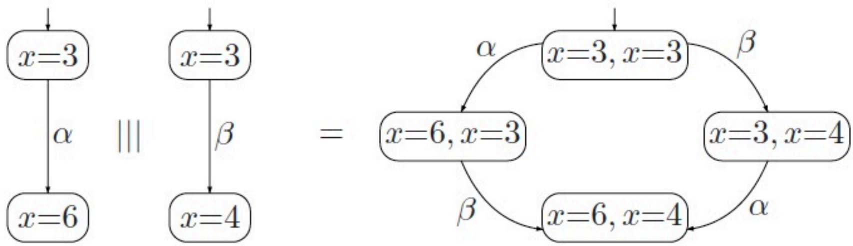
$$TS(PG_1) \mid\mid\mid TS(PG_2)$$

faithfully describes the concurrent behaviour of PG_1 and PG_2

what if they have variables in common?

Shared Variables

$$\underbrace{x := 2{\cdot}x}_{\text{action } \alpha} \;|||\; \underbrace{x := x+1}_{\text{action } \beta} \qquad \text{with initially } x = 3$$



$\langle x{=}6, x{=}4 \rangle$ is an inconsistent state!

$\Rightarrow$ this is not a faithful model of the concurrent execution of α and β.

Modelling Multi-threaded Program Graphs

- If PG_1 and PG_2 share no variables:

$$TS(PG_1) \; ||| \; TS(PG_2)$$

interleaving of transition systems

- If PG_1 and PG_2 share some variables:

$$TS(PG_1 \; ||| \; PG_2)$$

interleaving of program graphs (defined next)

- In general: $TS(PG_1) \; ||| \; TS(PG_2) \neq TS(PG_1 \; ||| \; PG_2)$

Interleaving of Program Graphs

Definition: Interleaving of program graphs

Let $PG_i = \left(Loc_i, Act_i, Effect_i, \longrightarrow_i, Loc_{0,i}, g_{0,i}\right)$ over variables Var_i, for i=1, 2.

Program graph $PG_1 \,|||\, PG_2$ over $Var_1 \cup Var_2$ is defined by:

$$\left(Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \longrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2}\right)$$

where $\longrightarrow$ is defined by the inference rules:

$$\frac{\ell_1 \xrightarrow{g:\alpha}_1 \ell_1'}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha} \langle \ell_1', \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \xrightarrow{g:\alpha}_2 \ell_2'}{\langle \ell_1, \ell_2 \rangle \xrightarrow{g:\alpha} \langle \ell_1, \ell_2' \rangle}$$

and $Effect(\alpha, \eta) \;=\; Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$.

A Toy Example

$$\underbrace{x := 2{\cdot}x}_{\text{action } \alpha} \;\;|||\;\; \underbrace{x := x + 1}_{\text{action } \beta} \qquad \text{with initially } x = 3$$

An Example with Two Threads: Dijkstra's Mutual Exclusion

Pseudo-code thread i=0:

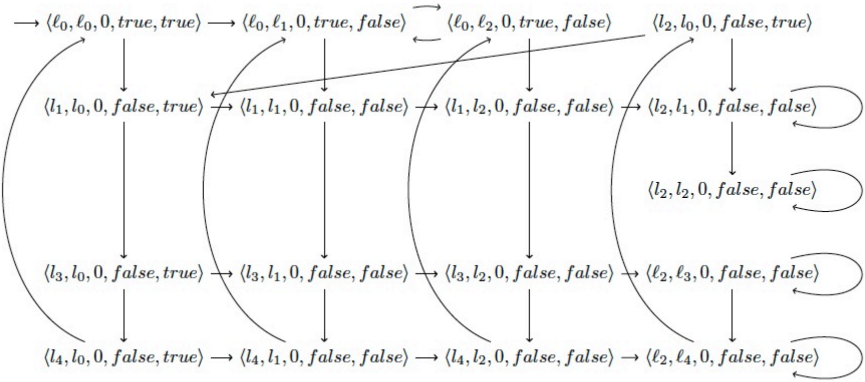
```
     int k := 0;
     b := [true, true];
ℓ0 { while (true) do
       b[i] := false;
ℓ1 {   while (k != i) do
ℓ2 {     while (not b[1-i]) do
           k := i;
         end
       end
ℓ3 {   critical_section;
ℓ4 {   b[i] := true;
     end
```

$|||$

Pseudo-code thread i=1:

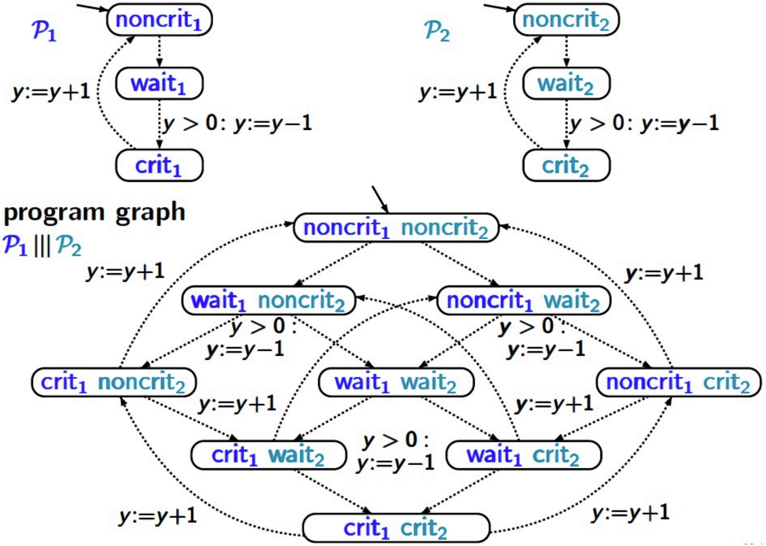
```
     int k := 0;
     b := [true, true];
ℓ0 { while (true) do
       b[i] := false;
ℓ1 {   while (k != i) do
ℓ2 {     while (not b[1-i]) do
           k := i;
         end
       end
ℓ3 {   critical_section;
ℓ4 {   b[i] := true;
     end
```

The Transition System: Dijkstra's Mutual Exclusion



We treated the states $\langle \ell_i, \ell_j, 0, b[0], b[1] \rangle$ and $\langle \ell_j, \ell_i, 1, b[1], b[0] \rangle$ as equivalent so as to reduce the size of the transition system.

Mutual Exclusion with Semaphores



On Atomicity

$$\underbrace{x := x + 1; y := 2x + 1; z := y \ \textbf{div} \ x}_{\text{non-atomic}} \ ||| \ {\color{red} x := 0}$$

Possible execution fragment:

$$\langle x = 11 \rangle \xrightarrow{x:=x+1} \langle x = 12 \rangle \xrightarrow{y:=2x+1} \langle x = 12 \rangle \xrightarrow{{\color{red} x:=0}} \langle x = 0 \rangle \xrightarrow{z:=y/x} \dagger \ldots$$

$$\underbrace{\langle x := x + 1; y := 2x + 1; z := y \ \textbf{div} \ x \rangle}_{\text{atomic}} \ ||| \ {\color{red} x := 0}$$

Either the left thread or the right thread is completed first:

$$\langle x = 11 \rangle \xrightarrow{x:=x+1} \langle x = 12 \rangle \xrightarrow{y:=2x+1} \langle x = 12 \rangle \xrightarrow{z:=y/x} \langle x = 12 \rangle \xrightarrow{{\color{red} x:=0}} \langle x = 0 \rangle$$

Peterson's Algorithm

P_1 **loop forever**

$\vdots$ (* non-critical actions *)

$\langle b_1 := \text{true};\ x := 2 \rangle;$ (* request *)

wait until $(x = 1 \ \lor\ \neg b_2)$

do critical section **od**

$b_1 := \text{false}$ (* release *)

$\vdots$ (* non-critical actions *)

end loop

b_i is true if and only if process P_i is waiting or in critical section

if both threads want to enter their critical section, x decides who gets access

Accessing a Bank Account

Thread Left behaves as follows:

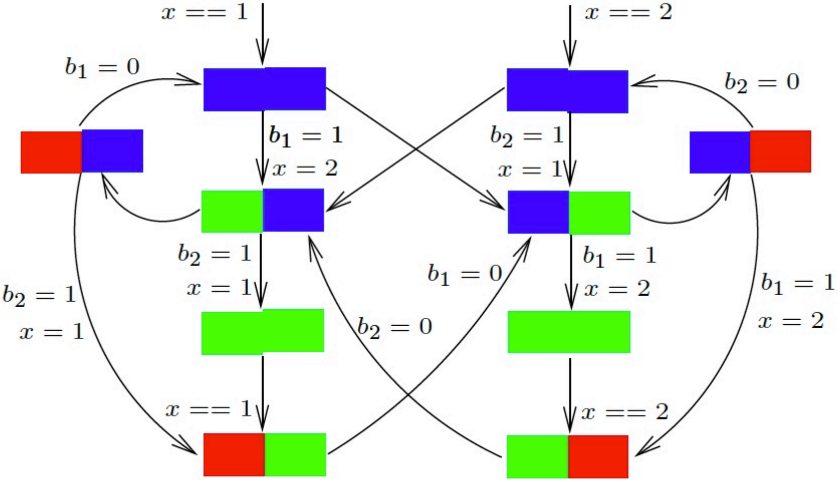
$$
\begin{array}{ll}
 & \textbf{while } \mathsf{true} \ \ \{ \\
 & \quad \ldots\ldots \\
nc: & \quad \langle b_1, x = \mathsf{true}, 2; \rangle \\
wt: & \quad \textbf{wait until}(x == 1 \,||\, \neg\, b_2)\, \{ \\
cs: & \quad\quad \ldots @\mathsf{account} \ldots \} \\
 & \quad b_1 = \mathsf{false}; \\
 & \quad \ldots\ldots \\
 & \}
\end{array}
$$

Thread Right behaves as follows:

$$
\begin{array}{ll}
 & \textbf{while } \mathsf{true} \ \ \{ \\
 & \quad \ldots\ldots \\
nc: & \quad \langle b_2, x = \mathsf{true}, 1; \rangle \\
wt: & \quad \textbf{wait until}(x == 2 \,||\, \neg\, b_1)\, \{ \\
cs: & \quad\quad \ldots @\mathsf{account} \ldots \} \\
 & \quad b_2 = \mathsf{false}; \\
 & \quad \ldots\ldots \\
 & \}
\end{array}
$$

Can we guarantee that only one thread at a time has access to the bank account?

The Transition System



Manual inspection reveals that mutual exclusion is guaranteed

A Non-Atomic Version

Thread Left behaves as follows:

	while true {
	......
nc :	$x = 2$;
rq :	b_1 = true;
wt :	**wait until**$(x == 1 \,\|\|\, \neg\, b_2)$ {
cs :	...@account...}
	b_1 = false;
	......
	}

Thread Right behaves as follows:

	while true {
	......
nc :	$x = 1$;
rq :	b_2 = true;
wt :	**wait until**$(x == 2 \,\|\|\, \neg\, b_1)$ {
cs :	...@account...}
	b_2 = false;
	......
	}

On Atomicity Again

Assume that the location inbetween the assignments $x := \ldots$ and $b_i :=$ true in program graph PG_i is called rq_i. Possible state sequence:

$$
\begin{array}{lllll}
\langle nc_1, & nc_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle nc_1, & rq_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle rq_1, & rq_2, & x = 2, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle wt_1, & rq_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle cs_1, & rq_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle cs_1, & wt_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle \\
\langle cs_1, & cs_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle
\end{array}
$$

This is a counterexample to the mutual exclusion property.

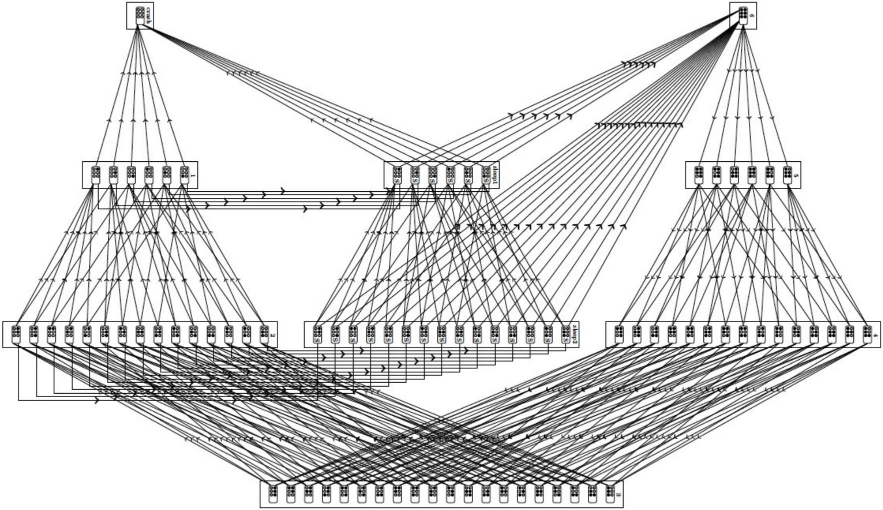
Overview

This Can be Modelled by Transition Systems Too

- Programs manipulating dynamic data structures
- Multi-threaded programs communicating via handshaking
- Multi-threaded programs communicating via unbounded buffers
- Multi-threaded programs using weak memory models
- Multi-threaded programs with fences (or: memory barriers)
- And many others

Overview

State Spaces Can Be Gigantic



A model of the Hubble telescope

Sequential Programs

- The $\#$ states of a program graph is worst case:

$$\left|\#\text{program locations}\right| \cdot \prod_{\text{variable } x} \left|\, dom(x) \,\right|$$

  - $\Rightarrow$ $\#$ states grows exponentially in the $\#$ program variables
  - N variables with k possible values each yields k^N states

- A program with 10 locations, 3 bools, 5 integers (in range $0 \ldots 9$):

$$10 \cdot 2^3 \cdot 10^5 \;=\; 800,000 \text{ states}$$

- Adding a single 50-positions bit-array yields $800,000 \cdot 2^{50}$ states

Multi-Threaded Programs

- The # states of $P_1 \, ||| \, \ldots \, ||| \, P_n$ is maximally:

$$\#\text{states of } P_1 \times \ldots \times \#\text{states of } P_n$$

$\Rightarrow$ # states grows exponentially in # threads

- The composition of N components of size k each yields k^N states

State Explosion Problem

The exponential growth of the state space in terms of the number of variables (as for program graphs) and number of threads (as for multi-threaded systems) gives rise to the state explosion problem.

In their basic form, model checking consists of enumerating and analysing the set of reachable states. Unfortunately, the number of states of even a relatively small system is often far greater than can be handled in a realistic computer.

Next Lecture

Thursday April 21, 12:30