

Model Checking
Lecture #2: Transition Systems
[Baier & Katoen, Chapter 2]

Joost-Pieter Katoen
Software Modeling and Verification Group

Model Checking Course, RWTH Aachen, WiSe 2019/2020

Introduction
Overview

What are Transition Systems?

Traces

Program Graphs

Multi-Threading

Other Forms of Concurrency

The State Explosion Problem

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

What are Transition Systems?

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

Traces

Program Graphs

Multi-Threading

Other Forms of Concurrency

The State Explosion Problem
Transition system

Definition: Transition system

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

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

\( S \) and \( \text{Act} \) are either finite or countably infinite.

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

A Mutual Exclusion Algorithm

![Diagram of a mutual exclusion algorithm]

For simplicity, actions are omitted in this example.

Direct Successors and Predecessors

\[
\begin{align*}
\text{Post}(s, \alpha) &= \left\{ s' \in S \mid s \xrightarrow{\alpha} s' \right\}, \\
\text{Post}(s) &= \bigcup_{\alpha \in \text{Act}} \text{Post}(s, \alpha) \\
\text{Pre}(s, \alpha) &= \left\{ s' \in S \mid s' \xrightarrow{\alpha} s \right\}, \\
\text{Pre}(s) &= \bigcup_{\alpha \in \text{Act}} \text{Pre}(s, \alpha). \\
\text{Post}(C, \alpha) &= \bigcup_{s \in C} \text{Post}(s, \alpha), \\
\text{Post}(C) &= \bigcup_{s \in C} \text{Post}(s) \text{ for } C \subseteq S. \\
\text{Pre}(C, \alpha) &= \bigcup_{s \in C} \text{Pre}(s, \alpha), \\
\text{Pre}(C) &= \bigcup_{s \in C} \text{Pre}(s) \text{ for } C \subseteq S.
\end{align*}
\]

State \( s \) is called terminal if and only if \( \text{Post}(s) = \emptyset \)

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 \( \alpha \) and set \( s = s' \)
  - od
**Executions**

**Definition: Executions**

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

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

  We also denote $\rho$ by: $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$.

- $\rho$ is maximal if $\rho$ is infinite or finite and ending in a terminal state.

- $\rho$ 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.

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

**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.
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 \( 2^{\mathit{AP}} \), i.e., they are in \( 2^{\mathit{AP}} \^\omega \)

\(^1\)This is an assumption commonly used throughout this lecture.

Example Traces

Consider the mutex transition system. Let \( \mathit{AP} = \{ \mathit{crit}_1, \mathit{crit}_2 \} \).

The trace of the path:

\[
\pi = \langle n_1, n_2, y = 1 \rangle \to \langle w_1, n_2, y = 1 \rangle \to \langle c_1, n_2, y = 0 \rangle \to \\
\langle n_1, n_2, y = 1 \rangle \to \langle n_1, w_2, y = 1 \rangle \to \langle n_1, c_2, y = 0 \rangle \to \ldots
\]

is:

\[
\text{trace}(\pi) = \Box \Box \{ \mathit{crit}_1 \} \Box \Box \{ \mathit{crit}_2 \} \Box \Box \{ \mathit{crit}_1 \} \Box \Box \{ \mathit{crit}_2 \} \ldots
\]

The finite trace of the finite path fragment:

\[
\bar{\pi} = \langle n_1, n_2, y = 1 \rangle \to \langle w_1, n_2, y = 1 \rangle \to \langle w_1, w_2, y = 1 \rangle \to \\
\langle w_1, c_2, y = 0 \rangle \to \langle w_1, n_2, y = 1 \rangle \to \langle c_1, n_2, y = 0 \rangle
\]

is:

\[
\text{trace}(\bar{\pi}) = \Box \Box \{ \mathit{crit}_2 \} \Box \{ \mathit{crit}_1 \}
\]

Overview

What are Transition Systems?

Traces

Program Graphs

Multi-Threaded

Other Forms of Concurrency

The State Explosion Problem
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 $\text{Var}$ be a collection of typed variables.

A program graph is a finite, rooted directed graph with:
- a finite set $\text{Loc}$ 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 $\alpha \in \text{Act}$, e.g., $x := x + 1$
  - an effect function describing the effect of an action on a variable valuation $\text{Var} \rightarrow \mathbb{R}$, e.g.,
    \[
    \text{Effect}(x := x + 1, [x \mapsto 5, y \mapsto 0]) = [x \mapsto 6, y \mapsto 0]
    \]
    $\eta(x) = 6, \eta(y) = 0$
    $\eta'(x) = 6, \eta'(y) = 0$
  - an initial Boolean condition, e.g., $x = 10 \land y < 3$

Program Graphs

Definition: Program graph

A program graph $\text{PG}$ over set $\text{Var}$ of typed variables is a tuple

\[
(\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)
\]

where

- $\text{Loc}$ is a set of locations with initial locations $\text{Loc}_0 \subseteq \text{Loc}$
- $\text{Act}$ is a set of actions
- $\text{Effect} : \text{Act} \times \text{Eval}(\text{Var}) \rightarrow \text{Eval}(\text{Var})$ is the effect function
- $\rightarrow \subseteq \text{Loc} \times \text{Cond}(\text{Var}) \times \text{Act} \times \text{Loc}$ is the edge relation
- $g_0 \in \text{Cond}(\text{Var})$ is the initial condition.

Example

Pseudo-code thread $i$:

\[
\begin{align*}
\text{int} \ k := 0; \\
\text{b} := [\text{true}, \text{true}]; \\
\ell_0 \{ \text{while (true) do} \\
\ell_1 \{ \text{while (k != i) do} \\
\ell_2 \{ \text{while (not b[i-j]) do} \\
\ell_3 \{ \text{critical_section;} \\
\ell_4 \{ \text{b[i] := true;} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end}
\end{align*}
\]

initially $\text{b} = [\text{true}, \text{true}]$
and $k = 0$
Program Graphs ⇔ Transition Systems

- Basic strategy: unfolding
  - state = location (current control) \( \ell \) + valuation \( \eta \)
  - initial state = initial location satisfying the initial condition

- Propositions and labelling
  - propositions: “at \( \ell \)” and “\( x \in D \)” for \( D \subseteq \text{dom}(x) \)
  - \( \langle \ell, \eta \rangle \) is labelled with “at \( \ell \)” and all conditions that hold in \( \eta \)

- If \( \ell \xrightarrow{\alpha} \ell' \) and \( g \) holds for the current valuation \( \eta \), then
  \[
  \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
  \]

Definition: Transition system of a program graph

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

\[
PG = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)
\]

over set \( \text{Var} \) of variables is the tuple \((S, \text{Act}, \rightarrow, I, \text{AP}, L)\) where

- \( S = \text{Loc} \times \text{Eval}(\text{Var}) \)
- \( \rightarrow \subseteq S \times \text{Act} \times S \) is defined by the rule:
  \[
  \ell \xrightarrow{\alpha} \ell' \land \eta \models g \\
  \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
  \]
- \( I = \{ \langle \ell, \eta \rangle \mid \ell \in \text{Loc}_0, \eta \models g_0 \} \)
- \( L(\ell, \eta) = \{ \ell \} \cup \{ g \in \text{Cond}(\text{Var}) \mid \eta \models g \} \).

Example

Pseudo-code thread \( i \):

```java
int k := 0;
b := [true, true];
\ell_0 \{ while (true) do
  b[i] := false;
\}
\ell_1 \{ while (k != i) do
  k := i;
\}
\ell_2 \{
  while (not b[i-1]) do
  k := i;
  end
end
\ell_3 \{ critical_section;
\ell_4 \{ b[i] := true;
  end
initially b = [true, true]
and k = 0
```

Overview

- What are Transition Systems?
- Traces
- Program Graphs
- Multi-Threading
- Other Forms of Concurrency
- The State Explosion Problem
Modelling Multi-Threading

- Transition systems
  - suited for modelling sequential data-dependent systems
  - 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 components

- Actions of independent components are merged or “interleaved”
  - a single processor is available
  - on which the actions of the processes are interlocked

- No assumptions are made on the order of processes
  - possible orders for non-terminating independent processes \( P \) and \( Q \):
    
    \[
    P \ Q \ P \ Q \ P \ Q \ P \ Q \ P \ Q \ P \ Q \ P \ Q \ P \ Q \ ...
    \]
    
    \[
    P \ P \ Q \ P \ P \ Q \ P \ P \ Q \ P \ P \ Q \ P \ P \ Q \ P \ ...
    \]
    
    \[
    P \ Q \ P \ P \ Q \ P \ P \ P \ Q \ P \ P \ P \ Q \ P \ P \ P \ Q \ P \ ...
    \]

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

Justification

\[
x := x + 1 \quad || \quad y := y - 2
\]

the effect of concurrently executed, independent actions \( \alpha \) and \( \beta \) 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

\[
TS_1 ||| TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)
\]

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

\[
\frac{s_1 \xrightarrow{\alpha} s'_1}{(s_1, s_2) \xrightarrow{\alpha} (s'_1, s_2)} \quad \text{and} \quad \frac{s_2 \xrightarrow{\beta} s'_2}{(s_1, s_2) \xrightarrow{\beta} (s_1, s'_2)}
\]
Interleaving of Program Graphs

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

$$TS(PG_1) \parallel TS(PG_2)$$

faithfully describes the concurrent behaviour of $PG_1$ and $PG_2$.

what if they have variables in common?

Modelling Multi-threaded Program Graphs

- If $PG_1$ and $PG_2$ share no variables:
  $$TS(PG_1) \parallel TS(PG_2)$$
  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: $TS(PG_1) ||| TS(PG_2) \neq TS(PG_1) ||| PG_2$

Definition: Interleaving of program graphs

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

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

$$(Loc_1 \times Loc_2, Act_1 \uplus Act_2, \rightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \land g_{0,2})$$

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

$$\frac{\ell_1 \xrightarrow{\alpha} \ell_1'} {\ell_1} \quad \text{and} \quad \frac{\ell_2 \xrightarrow{\alpha} \ell_2'} {\ell_2}$$

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

Shared Variables

$$x := \frac{2}{3} x \parallel x := x + 1$$

with initially $x = 3$

$$\begin{align*}
  x &= 3 \\
  \alpha \quad | \quad \beta \\
  x &= 6
\end{align*}$$

$(x=6, x=4)$ is an inconsistent state!

$\Rightarrow$ this is not a faithful mode of the concurrent execution of $\alpha$ and $\beta$. 

### A Toy Example

$x := 2 \cdot x$ \hspace{1cm} $x := x + 1$

**action $\alpha$**  \hspace{1cm} **action $\beta$**

with initially $x = 3$

### The Transition System

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

### On Atomicity

$\begin{align*}
x &:= x + 1; y := 2x + 1; z := y \div x \\
\end{align*}$

$\begin{align*}
x &:= 0
\end{align*}$

**non-atomic**

Possible execution fragment:

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

$\langle x := x + 1; y := 2x + 1; z := y \div x \rangle$

$\begin{align*}
x &:= 0
\end{align*}$

**atomic**

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

$\begin{align*}
\langle x = 11 \rangle \xrightarrow{x:=x+1} \langle x = 12 \rangle \xrightarrow{y:=2x+1} \langle x = 12 \rangle \xrightarrow{y=x/y} \langle x = 12 \rangle \xrightarrow{x:=0} \langle x = 0 \rangle
\end{align*}$
**Peterson’s Algorithm**

\[ P_1 \] loop forever 
\[ : \quad (* \text{non-critical actions} *) \]
\[ (b_1 := \text{true}; x := 2); \quad (* \text{request} *) \]
\[ \text{wait until} \ (x = 1 \lor \neg b_2) \]
\[ \text{do critical section} \quad \text{od} \]
\[ b_1 := \text{false} \quad (* \text{release} *) \]
\[ : \quad (* \text{non-critical actions} *) \]
\[ \text{end loop} \]

\( b_i \) is true if and only if process \( P_i \) is waiting or in critical section if both processes want to enter their critical section, \( x \) decides who gets access

---

**Accessing a Bank Account**

Person Left behaves as follows:

\[ \text{while true} \ { \}
\[ \quad \ldots \]
\[ \ 	ext{nc} : \quad (b_1, x = \text{true}, 2) \]
\[ \ 	ext{wt} : \quad \text{wait until}(x = 1 \lor \neg b_2) \}
\[ \ 	ext{cs} : \quad \ldots @\text{account} \ldots \]
\[ \quad b_1 = \text{false}; \]
\[ \quad \ldots \]
\[ \]}

Person Right behaves as follows:

\[ \text{while true} \ { \}
\[ \quad \ldots \]
\[ \ 	ext{nc} : \quad (b_2, x = \text{true}, 1) \]
\[ \ 	ext{wt} : \quad \text{wait until}(x = 2 \lor \neg b_1) \}
\[ \ 	ext{cs} : \quad \ldots @\text{account} \ldots \]
\[ \quad b_2 = \text{false}; \]
\[ \quad \ldots \]

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

---

**The Transition System**

Manual inspection reveals that mutual exclusion is guaranteed

---

**A Non-atomic Version**

Person Left behaves as follows:

\[ \text{while true} \ { \}
\[ \quad \ldots \]
\[ \ 	ext{nc} : \quad x = 2; \]
\[ \ 	ext{rq} : \quad b_1 = \text{true}; \]
\[ \ 	ext{wt} : \quad \text{wait until}(x = 1 \lor \neg b_2) \}
\[ \ 	ext{cs} : \quad \ldots @\text{account} \ldots \]
\[ \quad b_1 = \text{false}; \]
\[ \quad \ldots \]
\[ \]}

Person Right behaves as follows:

\[ \text{while true} \ { \}
\[ \quad \ldots \]
\[ \ 	ext{nc} : \quad x = 1; \]
\[ \ 	ext{rq} : \quad b_2 = \text{true}; \]
\[ \ 	ext{wt} : \quad \text{wait until}(x = 2 \lor \neg b_1) \}
\[ \ 	ext{cs} : \quad \ldots @\text{account} \ldots \]
\[ \quad b_2 = \text{false}; \]
\[ \quad \ldots \]

\[ ]}
On Atomicity Again

Assume that the location inbetween the assignments \( x := \ldots \) and \( b_i := \text{true} \) in program graph \( PG_j \) is called \( rq_i \). Possible state sequence:

\[
\langle nc_1, nc_2, x = 1, b_1 = \text{false}, b_2 = \text{false} \rangle
\]
\[
\langle nc_1, rq_2, x = 1, b_1 = \text{false}, b_2 = \text{false} \rangle
\]
\[
\langle rq_1, rq_2, x = 2, b_1 = \text{false}, b_2 = \text{false} \rangle
\]
\[
\langle wt_1, rq_2, x = 2, b_1 = \text{true}, b_2 = \text{false} \rangle
\]
\[
\langle cs_1, rq_2, x = 2, b_1 = \text{true}, b_2 = \text{false} \rangle
\]
\[
\langle cs_1, wt_2, x = 2, b_1 = \text{true}, b_2 = \text{true} \rangle
\]
\[
\langle cs_1, cs_2, x = 2, b_1 = \text{true}, b_2 = \text{true} \rangle
\]

This is a counterexample to the mutual exclusion property.

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
State Spaces Can Be Gigantic

A model of the Hubble telescope

Sequential Programs

- The # states of a program graph is worst case:
  \[ |\#\text{program locations}| \cdot \prod_{\text{variable } x} |\text{dom}(x)| \]
  \implies \text{# 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...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 \(P_1 \parallel \ldots \parallel P_n\) is maximally:
  \[ \#\text{states of } P_1 \times \ldots \times \#\text{states of } P_n \]
  \implies \text{# states grows exponentially in # components}

- 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 October 10, 10:30