

Model Checking

Transition Systems

[Baier & Katoen, Chapter 2] 

Joost-Pieter Katoen and Tim Quatmann

Software Modeling and Verification Group

RWTH Aachen, SoSe 2022

Overview

1 What are Transition Systems?

2 Traces

3 Program Graphs

4 Multi-Threading

5 Other Forms of Concurrency

6 The State Explosion Problem

Overview

1 What are Transition Systems?

2 Traces

3 Program Graphs

4 Multi-Threading

5 Other Forms of Concurrency

6 The State Explosion Problem

Transition systems

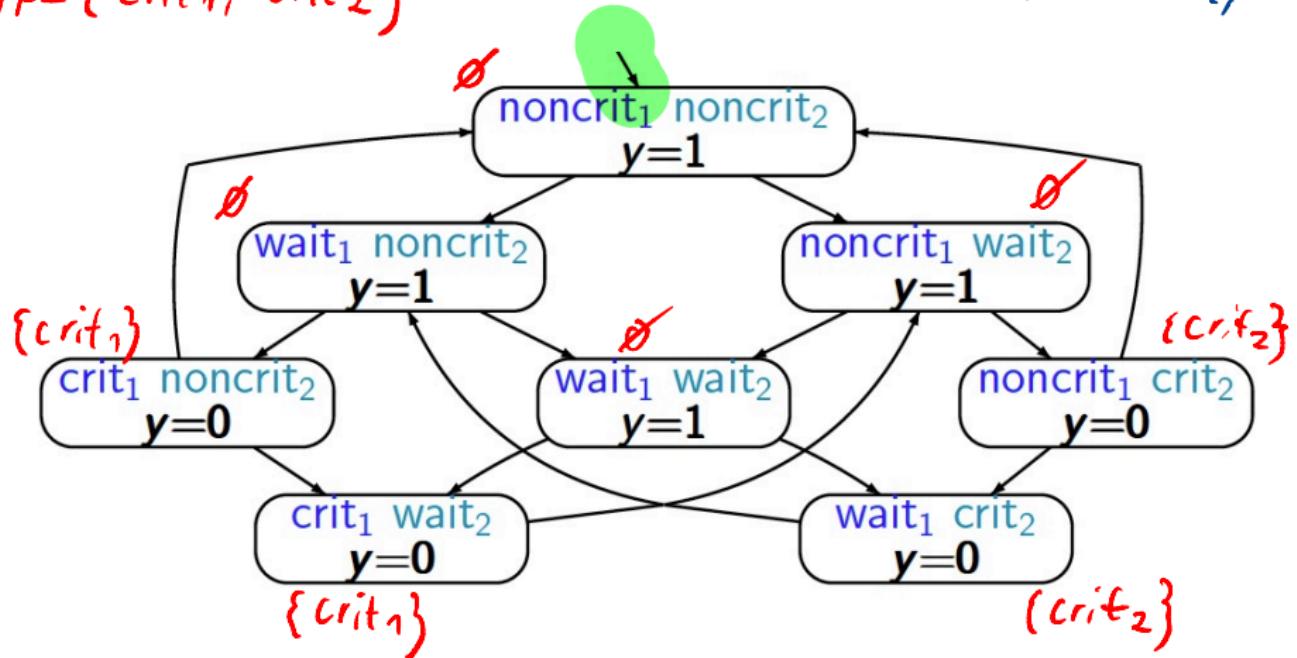
hardware sys.
software sys.

- ▶ Model to describe the behaviour of systems
directed graphs
- ▶ 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

$AP = \{crit_1, crit_2\}$

y is a binary semaphore
 $I := \{(noncrit_1, noncrit_2, y=1),$



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 finite or infinite
- ▶ Act is a set of actions source target
- ▶ $\rightarrow \subseteq S \times Act \times S$ is a transition relation
- ▶ $I \subseteq S$ is a set of initial states
- ▶ AP is a set of atomic propositions elementary formulae "x > 70"
- ▶ $L : S \rightarrow 2^{AP}$ is a labelling function

$(P(AP), \text{ Powerset of } AP)$

$L(s) \subseteq AP$

S and Act are either finite or countably infinite

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

Direct Successors and Predecessors

$$Post(s, \alpha) = \{ s' \in S \mid s \xrightarrow{\alpha} s' \}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

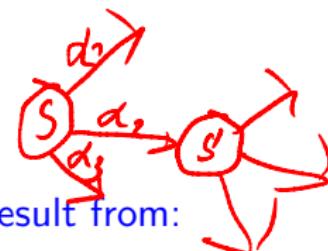
$$Pre(s, \alpha) = \{ s' \in S \mid s' \xrightarrow{\alpha} s \}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha).$$

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

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

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

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 α and set $s = s'$

od

Executions

$\Sigma^* = \text{set of finite words over } \Sigma$
 $\Sigma^\omega = \text{set of infinite words over } \Sigma$

Definition: Executions

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

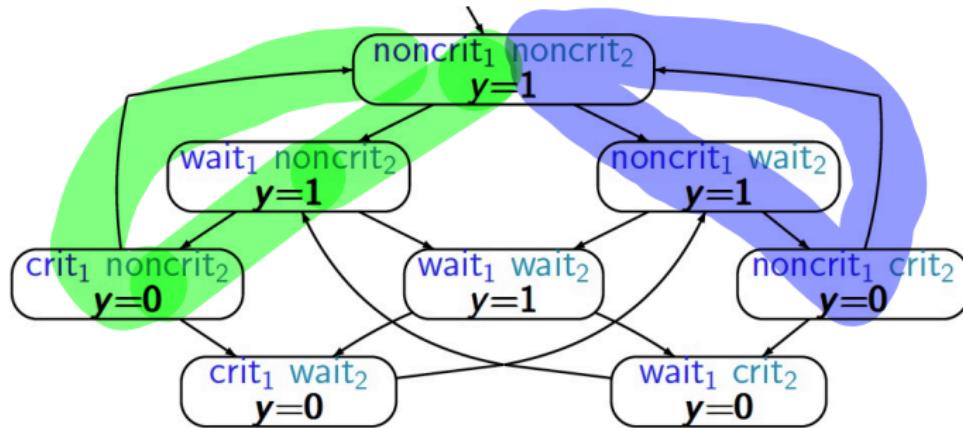
$$\rho = s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \dots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i.$$

We also denote ρ by: $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \dots$

- 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 $\pi = s_0 s_1 s_2 \dots$ starting in $s_0 = s \in S$.

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

Example Executions

 $s \in S$ $s = \langle n_1, n_2, y \rangle$ 

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

$s_0 \ s_1 \ s_2 \dots$

- ▶ Traces are sequences of the form $L(s_0) L(s_1) L(s_2) \dots$ record the (sets of) atomic propositions along an execution
- ▶ For transition systems without terminal states¹:
traces are infinite words over the alphabet 2^{AP} , i.e., they are in $(2^{AP})^\omega$

$\textcircled{L} \ \epsilon_{\text{dead}}()$

¹This 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

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

is the **infinite word** $trace(\rho) = L(s_0) L(s_1) L(s_2) \dots$ over $(2^{AP})^\omega$.
 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:

$$\pi = \langle n_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle w_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle c_1, n_2, y = 0 \rangle \xrightarrow{\quad} \langle n_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle n_1, w_2, y = 1 \rangle \xrightarrow{\quad} \langle n_1, c_2, y = 0 \rangle \xrightarrow{\quad} \dots$$

is:

\emptyset \emptyset $\{crit_1\}$

\emptyset \emptyset $\{crit_2\}$

$$trace(\pi) = \emptyset \emptyset \{crit_1\} \emptyset \emptyset \{crit_2\} \emptyset \emptyset \{crit_1\} \emptyset \emptyset \{crit_2\} \dots$$

The finite trace of the finite path fragment:

$$\hat{\pi} = \langle n_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle w_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle w_1, w_2, y = 1 \rangle \xrightarrow{\quad} \langle w_1, c_2, y = 0 \rangle \xrightarrow{\quad} \langle w_1, n_2, y = 1 \rangle \xrightarrow{\quad} \langle c_1, n_2, y = 0 \rangle$$

is:

\emptyset \emptyset \emptyset

$\{crit_2\}$ \emptyset $\{crit_1\}$

$$trace(\hat{\pi}) = \emptyset \emptyset \emptyset \{crit_2\} \emptyset \{crit_1\}$$

Overview

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

int, bool

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., $x < 10$ "guard"
 - ▶ an action $\alpha \in \text{Act}$, e.g., $x := x+1$ "Statement"

Intuition: if $x < 10$ then $x := x+1$

- ▶ an **effect** function describing the effect of an action on a variable valuation $\eta : \text{Var} \rightarrow \mathbb{D}$, e.g.,

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

- ▶ 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

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

- ▶ 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
- ▶ $\rightarrow \subseteq Loc \times \underbrace{Cond(Var)}_{\text{Boolean conditions over } Var} \times Act \times Loc$ is the **edge relation**
- ▶ $g_0 \in Cond(Var)$ is the initial **condition**.

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

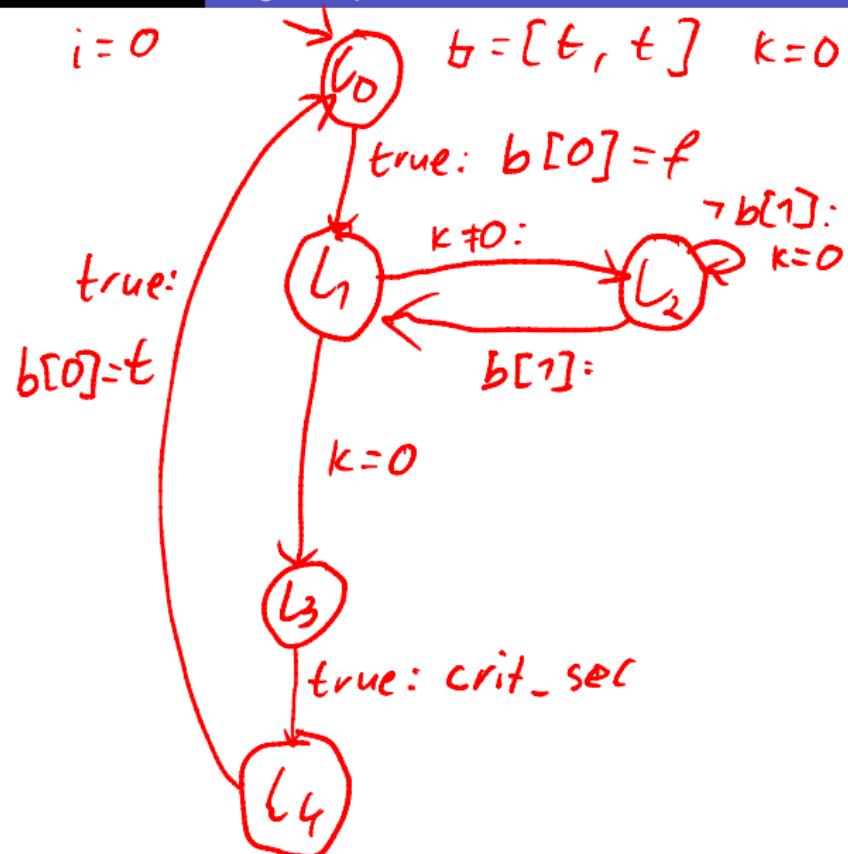
Example

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

```

int k := 0;
b := [true, true];
 $\ell_0 \left\{ \begin{array}{l} \text{while (true) do} \\ \quad b[i] := \text{false}; \end{array} \right.$ 
 $\ell_1 \left\{ \begin{array}{l} \text{while (k != i) do} \\ \quad \text{while (not } b[1-i]) \text{ do} \\ \quad \quad k := i; \\ \quad \text{end} \\ \text{end} \end{array} \right.$ 
 $\ell_2 \left\{ \begin{array}{l} \text{critical\_section;} \\ \ell_3 \left\{ \begin{array}{l} b[i] := \text{true}; \\ \text{end} \end{array} \right. \end{array} \right.$ 

```



initially $b = [\text{true}, \text{true}]$

and $k = 0$

Program Graphs \mapsto Transition Systems

- ▶ Basic strategy: **unfolding**
 - ▶ state = location (current control) ℓ + valuation $\eta : \text{Var} \rightarrow \mathbb{D}$
 - ▶ initial state = initial location satisfying the initial condition

$$\begin{array}{c} "k \neq 2'" \\ "x \text{ is odd}" \end{array}$$
- ▶ Propositions and labelling
 - ▶ propositions: “at ℓ ” and “ $x \in D$ ” for $D \subseteq \text{dom}(x)$
 - ▶ $\langle \ell, \eta \rangle$ 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

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

Program Graphs \mapsto Transition Systems

Definition: Transition system of a program graph

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

$$PG = (Loc, \text{Act}, \text{Effect}, \rightarrow, Loc_0, g_0)$$

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

- ▶ $S = Loc \times Eval(Var)$ $s = \langle \ell, (x \mapsto 10, y \mapsto 7) \rangle$
 - ▶ $\rightarrow \subseteq S \times Act \times S$ is defined by the rule:
- $$\frac{\ell \xrightarrow{g:\alpha} \ell' \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle}$$
- premise
conclusion
- ▶ $I = \{ \langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0 \}$
 - ▶ $L(\langle \ell, \eta \rangle) = \{ \ell \} \cup \{ g \in Cond(Var) \mid \eta \models g \}$.

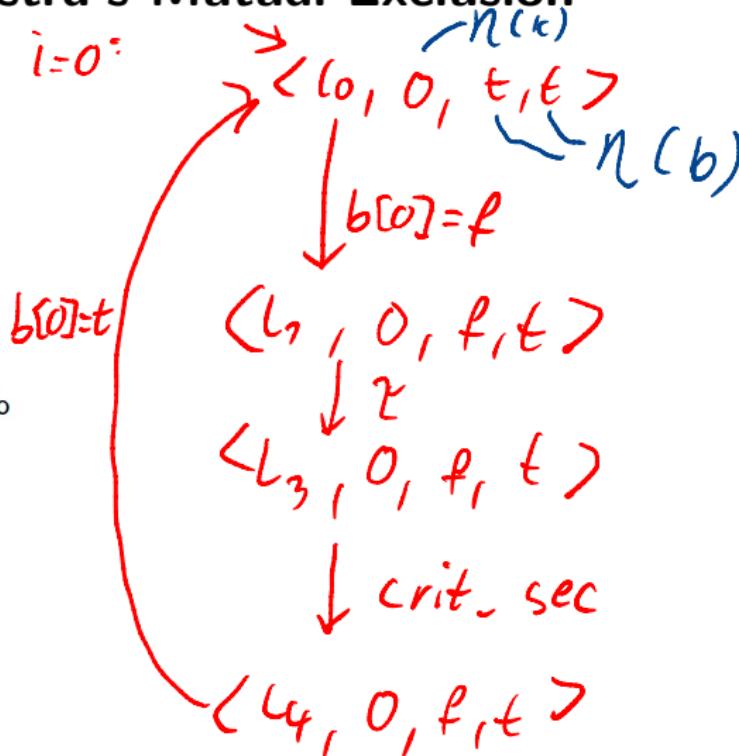
Example: Dijkstra's Mutual Exclusion

Pseudo-code thread i :

```

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

```



initially $b = [\text{true}, \text{true}]$
and $k = 0$

Overview

1 What are Transition Systems?

2 Traces

3 Program Graphs

4 Multi-Threading

5 Other Forms of Concurrency

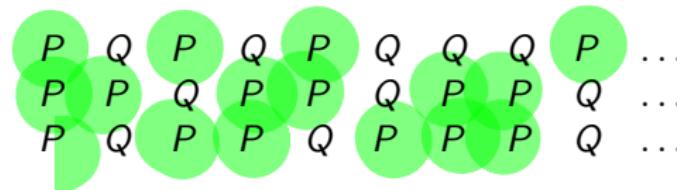
6 The State Explosion Problem

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? *Yes*

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 :



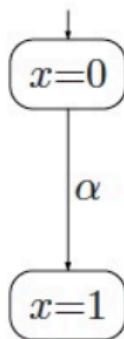
- ▶ assumption: there is a scheduler with an a priori **unknown** strategy

Justification

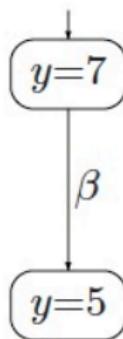
r "Parallel"

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

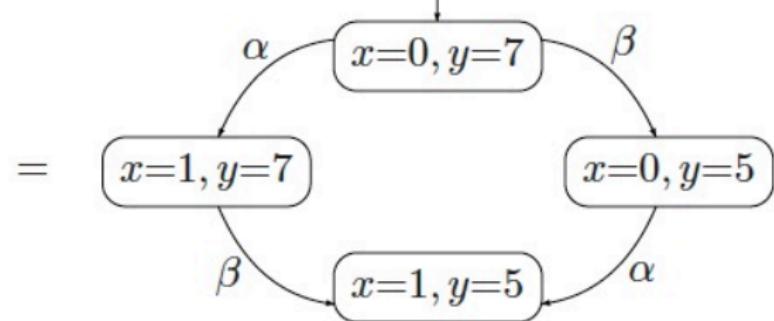
TS_1 :



TS_2 :



$TS_1 \parallel TS_2$



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

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

disjoint

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:

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


\rightarrow is the smallest relation sat.

Interleaving of Program Graphs

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

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

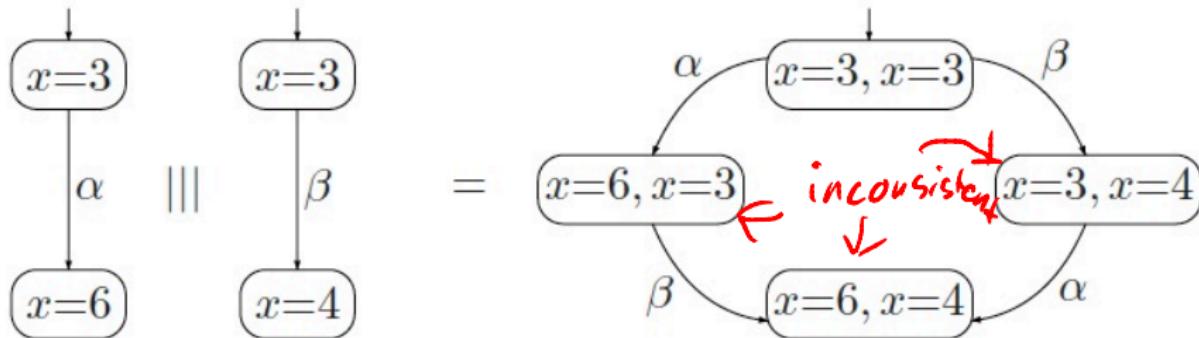
faithfully describes the concurrent behaviour of PG_1 and PG_2

what if they have variables in common?

Shared Variables

$$\underbrace{x := 2 \cdot x}_{\text{action } \alpha} \parallel \underbrace{x := x + 1}_{\text{action } \beta}$$

with initially $x = 3$



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

⇒ 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:

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

interleaving of transition systems

- ▶ If PG_1 and PG_2 **share some variables**:

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

interleaving of program graphs (defined next)

- ▶ In general: $TS(PG_1) \parallel\!\!\parallel TS(PG_2) \neq TS(PG_1 \parallel\!\!\parallel PG_2)$

Interleaving of Program Graphs

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

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

where \rightarrow is defined by the inference rules:

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

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

A Toy Example

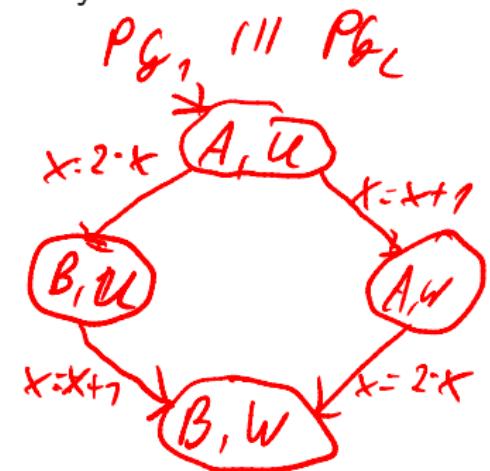
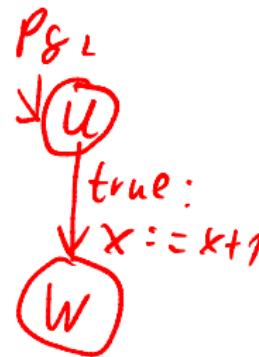
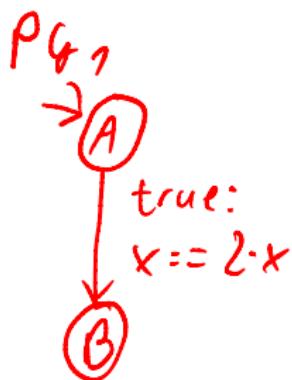
$TSL(PG_1, \parallel PG_2)$

$\rightarrow (A, U), 3$

$x := x + 1 \xrightarrow{} A, W, 4 \xrightarrow{x = 2 \cdot x} B, W, 8$

$x := 2 \cdot x \xrightarrow{} B, U, 6 \xrightarrow{x = x + 1} B, W, 2$

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



An Example with Two Threads: Dijkstra's Mutual Exclusion

Pseudo-code thread $i=0$:

```

int k := 0;
b := [true, true];
 $\ell_0 \left\{ \begin{array}{l} \text{while (true) do} \\ \quad b[0] := \text{false}; \end{array} \right.$ 
 $\ell_1 \left\{ \begin{array}{l} \text{while (k != } \textcolor{red}{0} \text{) do} \\ \quad \text{while (not b[1]) do} \\ \quad \quad k := \textcolor{red}{0}; \\ \quad \text{end} \end{array} \right.$ 
 $\ell_2 \left\{ \begin{array}{l} \text{end} \\ \text{end} \end{array} \right.$ 
 $\ell_3 \left\{ \begin{array}{l} \text{critical\_section;} \end{array} \right.$ 
 $\ell_4 \left\{ \begin{array}{l} b[i] := \text{true;} \\ \text{end} \end{array} \right.$ 

```

|||

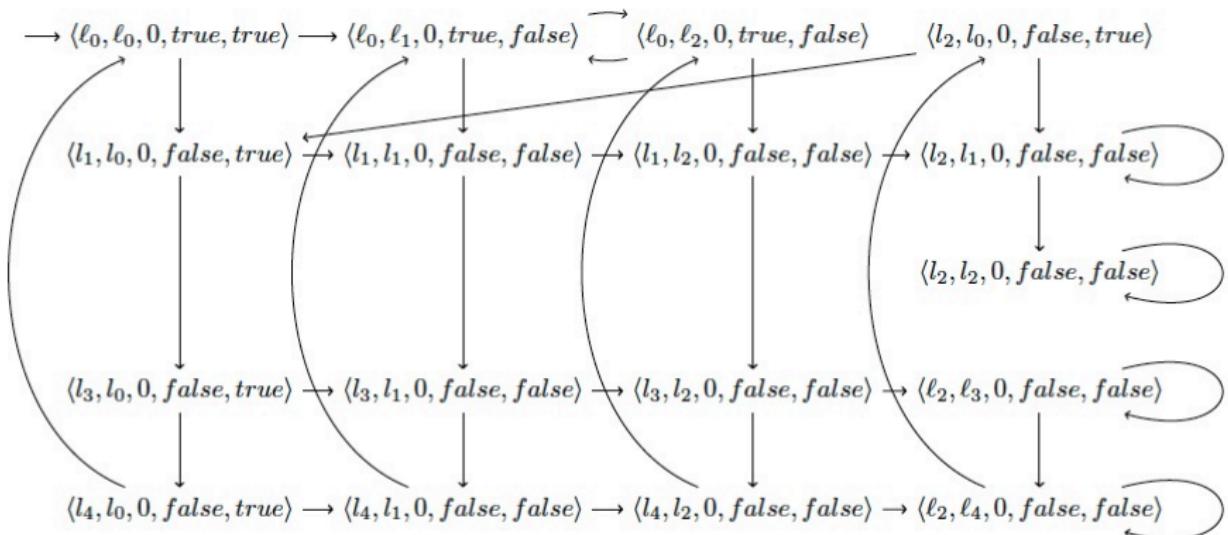
Pseudo-code thread $i=1$:

```

int k := 0;
b := [true, true];
 $\ell_0 \left\{ \begin{array}{l} \text{while (true) do} \\ \quad b[1] := \text{false}; \end{array} \right.$ 
 $\ell_1 \left\{ \begin{array}{l} \text{while (k != } \textcolor{red}{1} \text{) do} \\ \quad \text{while (not b[0]) do} \\ \quad \quad k := \textcolor{red}{1}; \\ \quad \text{end} \end{array} \right.$ 
 $\ell_2 \left\{ \begin{array}{l} \text{end} \\ \text{end} \end{array} \right.$ 
 $\ell_3 \left\{ \begin{array}{l} \text{critical\_section;} \end{array} \right.$ 
 $\ell_4 \left\{ \begin{array}{l} b[i] := \text{true;} \\ \text{end} \end{array} \right.$ 

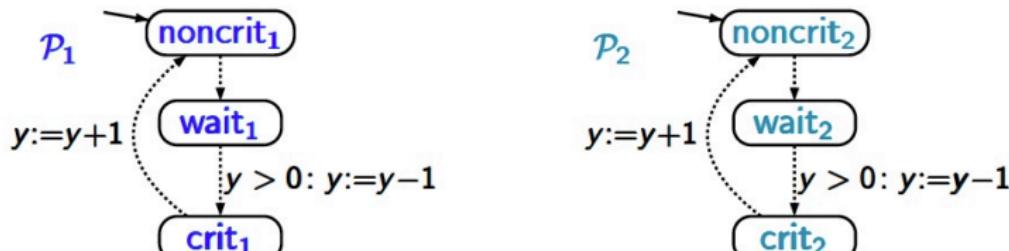
```

The Transition System: Dijkstra's Mutual Exclusion



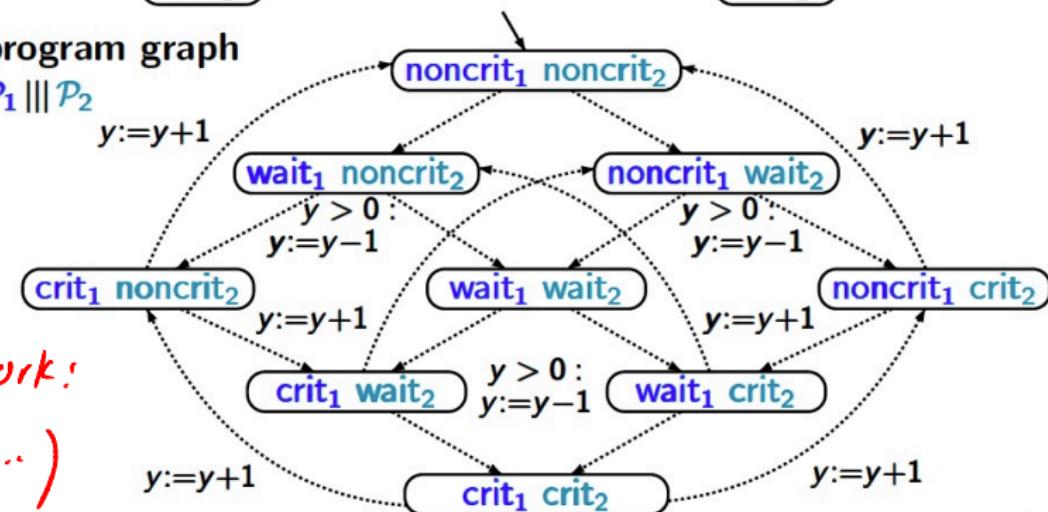
We treated the states $\langle \ell_i, \ell_j, 0, b[0], b[1] \rangle$ and $\langle \ell_j, \ell_i, 1, b[1], b[0] \rangle$ as equivalent so as to reduce the size of the transition system.

Mutual Exclusion with Semaphores



program graph

$P_1 \parallel P_2$



Homework!

TS(...)

On Atomicity

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

Possible execution fragment:

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

division
by 0

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

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

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

Peterson's Algorithm

```
P1 loop forever
  :
  (* non-critical actions *)
  ⟨b1 := true; x := 2⟩; -atomic (* request *)
  wait until (x = 1 ∨ ¬b2)
  do critical section od
  b1 := false (* release *)
  :
  (* 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:

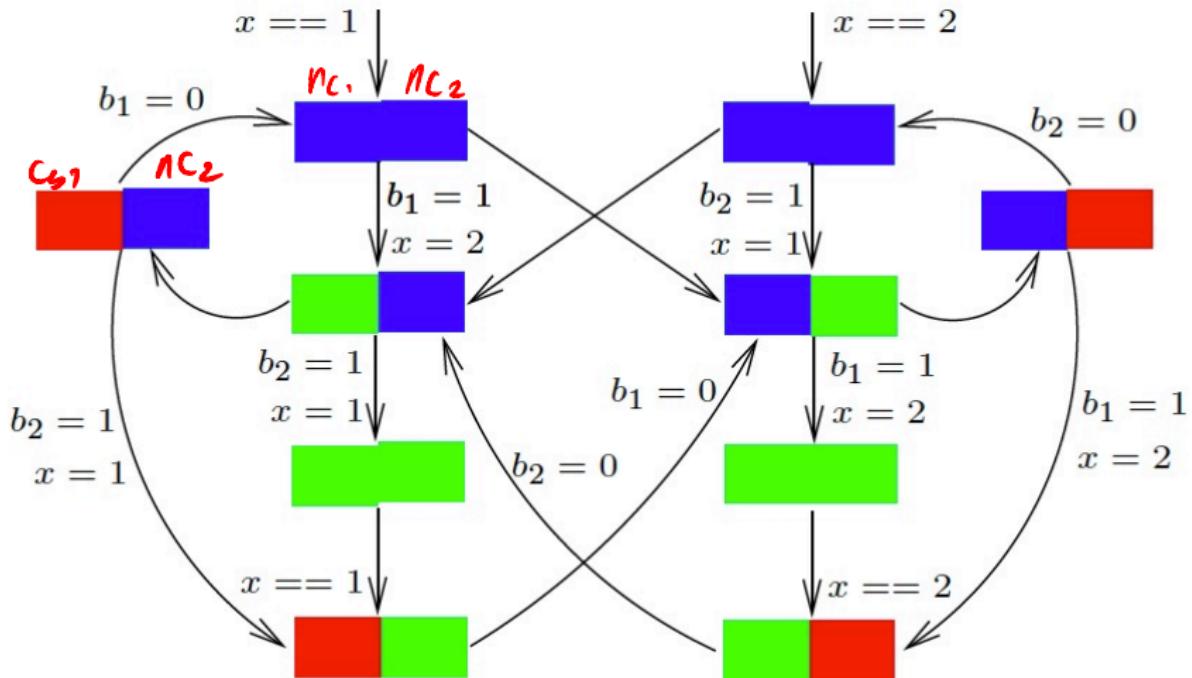
```
while true {  
    .....  
    nc :   ⟨ $b_1$ ,  $x = \text{true}$ , 2; ⟩  
    wt :   wait until( $x == 1 \parallel \neg b_2$ ) {  
    cs :       ... @account ... }  
     $b_1 = \text{false};$   
    .....  
}
```

Thread Right behaves as follows:

```
while true {  
    .....  
    nc :   ⟨ $b_2$ ,  $x = \text{true}$ , 1; ⟩  
    wt :   wait until( $x == 2 \parallel \neg b_1$ ) {  
    cs :       ... @account ... }  
     $b_2 = \text{false};$   
    .....  
}
```

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 : b1 = true;  
    wt : wait until(x == 1 || !b2) {  
        cs : ... @account ...}  
        b1 = false;  
    .....  
}
```

Thread Right behaves as follows:

```
while true {  
    .....  
    nc : x = 1;  
    rq : b2 = true;  
    wt : wait until(x == 2 || !b1) {  
        cs : ... @account ...}  
        b2 = false;  
    .....  
}
```

On Atomicity Again

Assume that the location inbetween the assignments $x := \dots$ and $b_i := \text{true}$ in program graph PG_i 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.

Overview

1 What are Transition Systems?

2 Traces

3 Program Graphs

4 Multi-Threading

5 Other Forms of Concurrency

6 The State Explosion Problem

This Can be Modelled by Transition Systems Too

e.g. balanced Tree

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

Overview

1 What are Transition Systems?

2 Traces

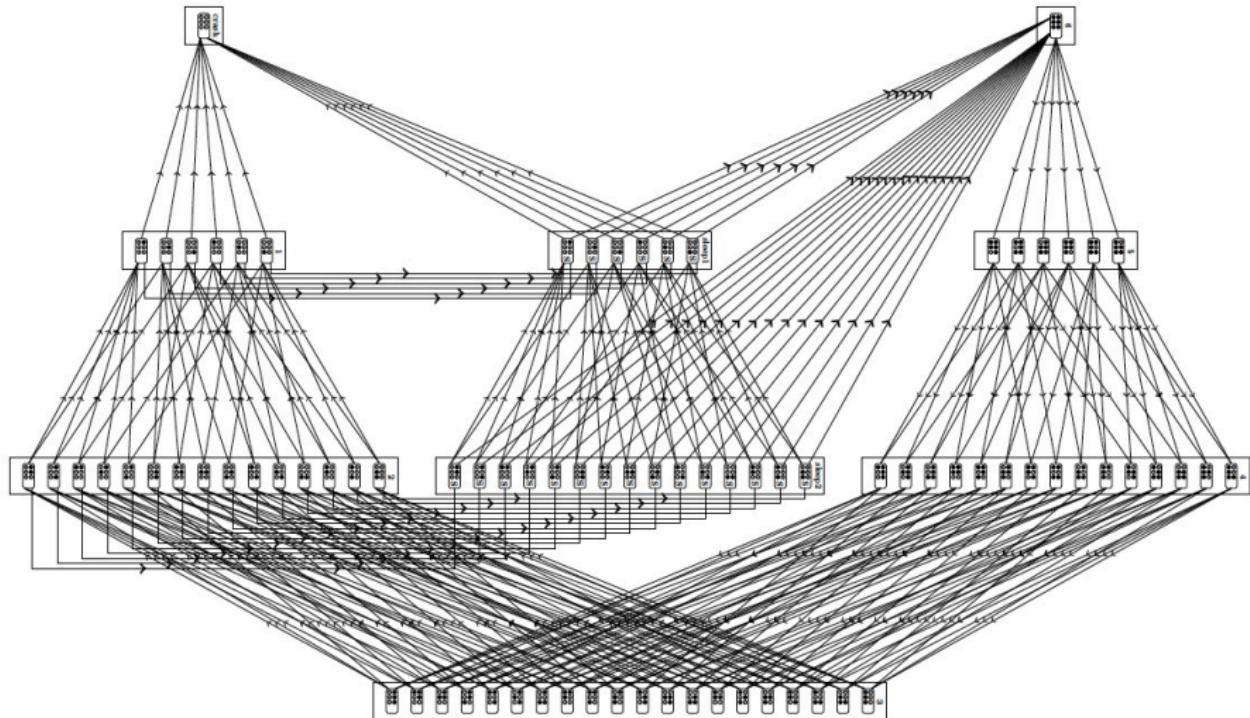
3 Program Graphs

4 Multi-Threading

5 Other Forms of Concurrency

6 The State Explosion Problem

State Spaces Can Be Gigantic



A model of the Hubble telescope

Sequential Programs

nr. of

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

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

variable valuations

⇒ # 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
 $\approx 10^{21}$

Multi-Threaded Programs

- ▶ The # states of $P_1 \parallel \dots \parallel P_n$ is maximally:

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

⇒ # 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

≈ 1.5 weeks