Verification

Lecture 3

Bernd Finkbeiner
Peter Faymonville
Michael Gerke

UNIVERSITÄT
DES
SAARLANDES

REVIEW: Channel systems

A program graph over $(Var, Chan)$ is a tuple

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

where

$$\to \ \subseteq \ Loc \times (Cond(Var) \times Act) \times Loc \ \cup \ \underbrace{Loc \times Comm \times Loc}_{\text{communication actions}}$$

A channel system CS over $(\bigcup_{0<i\leq n} Var_i, Chan)$:

$$CS = [PG_1 \mid \ldots \mid PG_n]$$

with program graphs PG_i over $(Var_i, Chan)$

REVIEW: Transition system semantics of a channel system

Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$ with

$$PG_i = (Loc_i, Act_i, Effect_i, \rightsquigarrow_i, Loc_{0,i}, g_{0,i}), \qquad \text{for } 0 < i \leq n$$

$TS(CS)$ is the transition system $(S, Act, \rightarrow, I, AP, L)$ where:

- $S = (Loc_1 \times \cdots \times Loc_n) \times Eval(Var) \times Eval(Chan)$
- $Act = (\biguplus_{0<i\leq n} Act_i) \uplus \{ \tau \}$
- $\rightarrow$ is defined by the inference rules on the next slides
- $I = \left\{ \langle \ell_1, \ldots, \ell_n, \eta, \xi_0 \rangle \mid \forall i.\ (\ell_i \in Loc_{0,i} \ \& \ \eta \models g_{0,i}) \ \& \ \forall c.\ \xi_0(c) = \varepsilon \right\}$
- $AP = \biguplus_{0<i\leq n} Loc_i \uplus Cond(Var)$
- $L(\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle) = \{ \ell_1, \ldots, \ell_n \} \cup \{ g \in Cond(Var) \mid \eta \models g \}$

REVIEW: Inference rules (I)

- Interleaving for $\alpha \in Act_i$:

$$\frac{\ell_i \xrightarrow{g:\alpha} \ell_i' \quad \wedge \quad \eta \vDash g}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = Effect(\alpha, \eta)$

- Synchronous message passing over $c \in Chan$, $cap(c) = 0$:

$$\frac{\ell_i \xrightarrow{c?x} \ell_i' \quad \wedge \quad \ell_j \xrightarrow{c!v} \ell_j' \quad \wedge \quad i \neq j}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \eta[x := v]$.

REVIEW: Inference rules (II)

- Asynchronous message passing for $c \in Chan$, $cap(c) > 0$:
  - receive a value along channel c and assign it to variable x:

$$\frac{\ell_i \xrightarrow{c?x} \ell_i' \quad \wedge \quad len(\xi(c)) = k > 0 \quad \wedge \quad \xi(c) \;=\; v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi' \rangle}$$

where $\eta' = \eta[x := v_1]$ and $\xi' = \xi[c := v_2 \ldots v_k]$.

  - transmit value $v \in dom(c)$ over channel c:

$$\frac{\ell_i \xrightarrow{c!v} \ell_i' \quad \wedge \quad len(\xi(c)) = k < cap(c) \quad \wedge \quad \xi(c) \;=\; v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi' \rangle}$$

where $\xi' = \xi[c := v_1\, v_2 \ldots v_k\, v]$.

REVIEW: nanoPromela

nanoPromela-program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ with $\mathcal{P}_i$ processes
A process is specified by a statement:

$$
\begin{array}{lcl}
\text{stmt} & ::= & \texttt{skip} \;\big|\; x := \text{expr} \;\big|\; c?x \;\big|\; c!\text{expr} \;\big| \\
& & \text{stmt}_1 \,;\, \text{stmt}_2 \;\big|\; \texttt{atomic}\{\text{assignments}\} \;\big| \\
& & \textbf{if} \quad :: g_1 \Rightarrow \text{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \text{stmt}_n \quad \textbf{fi} \quad \big| \\
& & \textbf{do} \quad :: g_1 \Rightarrow \text{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \text{stmt}_n \quad \textbf{od} \\
\text{assignments} & ::= & x_1 := \text{expr}_1 \,;\, x_2 := \text{expr}_2 \,;\, \ldots \,;\, x_m := \text{expr}_m
\end{array}
$$

x is a variable in *Var*, expr an expression and c a channel, g_i a guard

assume the Promela specification is type-consistent

REVIEW: Peterson's algorithm

The nanoPromela-code of process $\mathcal{P}_1$ is given by the statement:

$$
\begin{array}{lll}
\textbf{do} & :: \ \text{true} \Rightarrow & \texttt{skip}; \\
 & & \texttt{atomic}\{b_1 := \text{true}; x := 2\}; \\
 & & \textbf{if} \ :: \ (x = 1) \vee \neg b_2 \ \Rightarrow \ \mathit{crit}_1 := \text{true} \ \textbf{fi} \\
 & & \texttt{atomic}\{\mathit{crit}_1 := \text{false}; b_1 := \text{false}\} \\
\textbf{od} & &
\end{array}
$$

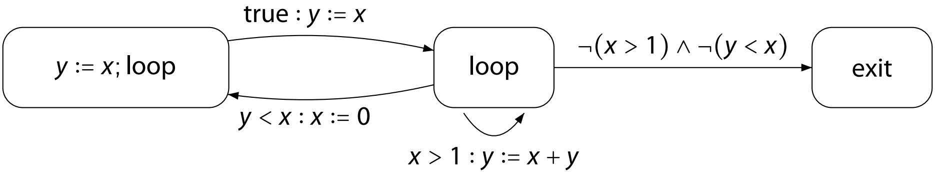
Formal semantics

The semantics of a nanoPromela-statement over $(Var, Chan)$ is a program graph over $(Var, Chan)$.

The program graphs $PG_1, \ldots, PG_n$ for the processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of a nanoPromela-program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ constitute a channel system over $(Var, Chan)$

Example

$$
\begin{array}{llllll}
\text{loop} & = & \textbf{do} & :: & x > 1 & \Rightarrow & y := x + y \\
 & & & :: & y < x & \Rightarrow & x := 0;\ y := x \\
 & & \textbf{od} & & & &
\end{array}
$$



Substatements

- substatements: potential locations of intermediate states during the execution of a statement.
- for stmt $\in \{\texttt{skip}, x := \text{expr}, c?x, c!\text{expr}\}$:
  $sub(\text{stmt}) = \{\text{stmt}, \text{exit}\}$
- $sub(\text{stmt}_1 ; \text{stmt}_2) =$
  $\{stmt' ; stmt_2 \mid \text{stmt}' \in sub(\text{stmt}_1) \setminus \{\text{exit}\}\} \cup sub(\text{stmt}_2)$
- for cond_cmd $=$ $\textbf{if} :: g_1 \Rightarrow \text{stmt}_1 \ldots :: g_n \Rightarrow \text{stmt}_n \textbf{ fi}$,
  $sub(\text{cond\_cmd}) = \{\text{stmt}\} \cup \bigcup_{1 \leq i \leq n} sub(\text{stmt}_i)$
- for loop $=$ $\textbf{do} :: g_1 \Rightarrow \text{stmt}_1 \ldots :: g_n \Rightarrow \text{stmt}_n \textbf{ od}$,
  $sub(\text{loop}) = \{\text{loop}, \text{exit}\} \cup$
  $$\bigcup_{1 \leq i \leq n} \{\text{stmt}' ; \text{loop} \mid \text{stmt}' \in sub(\text{stmt}_i) \setminus \{\text{exit}\}\}$$

Inference rules

$$\frac{}{\texttt{skip} \xrightarrow{\text{true: } id} \text{exit}}$$

where *id* denotes an action that does not change the values of the variables

$$\frac{}{x := \text{expr} \xrightarrow{\text{true} : \text{assign}(x, \text{expr})} \text{exit}}$$

$\text{assign}(x, \text{expr})$ denotes the action that only changes x, no other variables

$$\frac{}{c?x \xrightarrow{c?x} \text{exit}} \qquad \frac{}{c!\text{expr} \xrightarrow{c!\text{expr}} \text{exit}}$$

Inference rules

$$\frac{}{\texttt{atomic}\{x_1 := \mathrm{expr}_1; \ldots; x_m := \mathrm{expr}_m\} \xrightarrow{\mathrm{true} : \alpha_m} \mathrm{exit}}$$

where $\alpha_0 = \mathit{id}$, $\alpha_i = \mathit{Effect}(\mathrm{assign}(x_i, \mathrm{expr}_i), \mathit{Effect}(\alpha_{i-1}, \eta))$ for $1 \leq i \leq m$

$$\frac{\mathrm{stmt}_1 \xrightarrow{g:\alpha} \mathrm{stmt}_1' \neq \mathrm{exit}}{\mathrm{stmt}_1; \mathrm{stmt}_2 \xrightarrow{g:\alpha} \mathrm{stmt}_1'; \mathrm{stmt}_2}$$

$$\frac{\mathrm{stmt}_1 \xrightarrow{g:\alpha} \mathrm{exit}}{\mathrm{stmt}_1; \mathrm{stmt}_2 \xrightarrow{g:\alpha} \mathrm{stmt}_2}$$

Inference rules

$$\frac{\mathsf{stmt}_i \xrightarrow{h:\alpha} \mathsf{stmt}'_i}{\mathsf{cond\_cmd} \xrightarrow{g_i \wedge h:\alpha} \mathsf{stmt}'_i}$$

$$\frac{\mathsf{stmt}_i \xrightarrow{h:\alpha} \mathsf{stmt}'_i \neq \mathsf{exit}}{\mathsf{loop} \xrightarrow{g_i \wedge h:\alpha} \mathsf{stmt}'_i ; \mathsf{loop}} \qquad \frac{\mathsf{stmt}_i \xrightarrow{h:\alpha} \mathsf{exit}}{\mathsf{loop} \xrightarrow{g_i \wedge h:\alpha} \mathsf{loop}}$$

$$\frac{}{\mathsf{loop} \xrightarrow{\neg g_1 \wedge \ldots \wedge \neg g_n} \mathsf{exit}}$$

The state-space explosion problem

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

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

  $\Rightarrow$ number of states grows exponentially in the number of 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

Concurrent programs

- The # states of $P \equiv P_1 \parallel \ldots \parallel P_n$ is maximally:

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

$\Rightarrow$ # states grows exponentially with the number of components

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

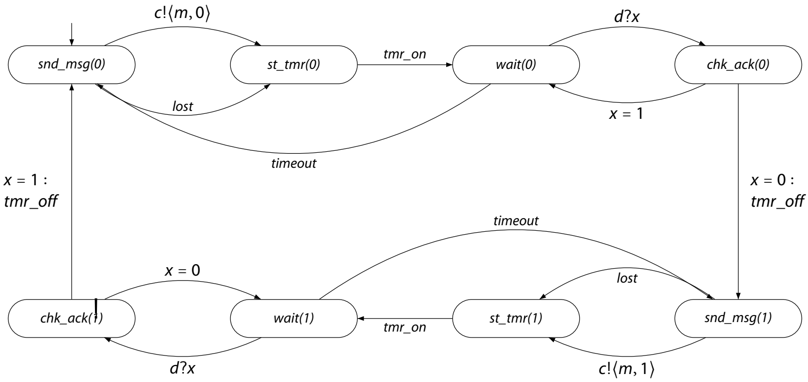
Channel systems

- Asynchronous communication of processes via channels
  - each channel c has a bounded capacity $cap(c)$
  - if a channel has capacity 0, we obtain handshaking
- $\#$ states of system with N components and K channels is:

$$\prod_{i=1}^{N}\left(|\#\text{program locations}| \prod_{\text{variable } x} |dom(x)|\right) \cdot \prod_{j=1}^{K} |dom(c_j)|^{cap(c_j)}$$

this is the underlying structure of Promela

The alternating bit protocol



channel capacity 10, and datums are bits,

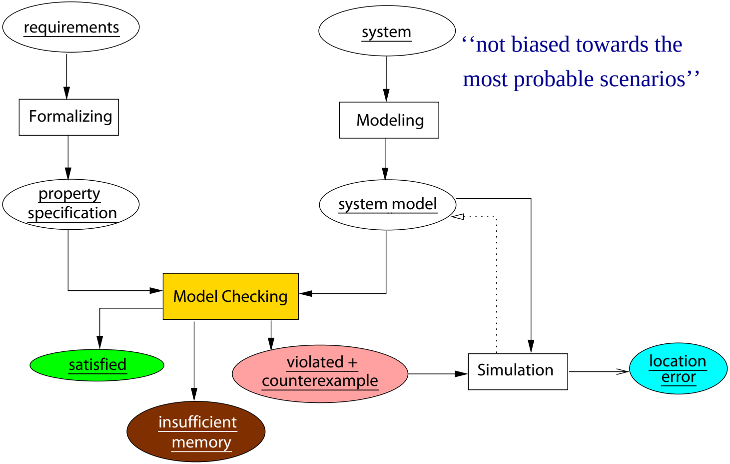
yields $2 \cdot 8 \cdot 6 \cdot 4^{10} \cdot 2^{10} = 3 \cdot 2^{35} \approx 10^{11}$ states

Summary: Transition Systems

- Transition systems are fundamental for modeling software and hardware
- Interleaving = execution of independent concurrent processes by nondeterminism
- For shared variable communication use composition on program graphs
- Handshaking on a set H of actions amounts to
  - executing action $\notin H$ autonomously (= interleaving)
  - those in H simultaneously
- Channel systems = program graphs + first-in first-out communication channels
  - handshaking for channels of capacity 0
  - asynchronous message passing when capacity exceeds 0
  - semantical model of Promela
- Size of transition systems grows exponentially
  - in the number of concurrent components and the number of variables

Linear-Time Properties

REVIEW: model checking



“not biased towards the most probable scenarios”

REVIEW: executions

- A finite execution fragment ρ of TS is an alternating sequence of states and actions ending with a state:

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

- An infinite execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

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

- An execution of TS is an initial, maximal execution fragment
  - a maximal execution fragment is either finite ending in a terminal state, or infinite
  - an execution fragment is initial if $s_0 \in I$

State graph

- The state graph of TS, notation $G(TS)$, is the digraph (V, E)

  with vertices $V = S$ and edges $E = \{(s, s') \in S \times S \mid s' \in Post(s)\}$

  $\Rightarrow$ omit all state and transition labels in TS and ignore being initial

- $Post^*(s)$ is the set of states reachable $G(TS)$ from s

$$Post^*(C) = \bigcup_{s \in C} Post^*(s) \quad \text{for } C \subseteq S$$

- The notations $Pre^*(s)$ and $Pre^*(C)$ have analogous meaning
- The set of reachable states: $Reach(TS) = Post^*(I)$

Path fragments

- A path fragment is an execution fragment without actions
- A finite path fragment $\widehat{\pi}$ of TS is a state sequence:

$$\widehat{\pi} = s_0 \, s_1 \ldots s_n \quad \text{such that} \quad s_{i+1} \in Post(s_i) \text{ for all } 0 \leq i < n \text{ where } n \geq 0$$

- An infinite path fragment π of TS is an infinite state sequence:

$$\pi = s_0 \, s_1 \, s_2 \ldots \quad \text{such that } s_{i+1} \in Post(s_i) \text{ for all } i \geq 0$$

- A path of TS is an initial, maximal path fragment
  - a maximal path fragment is either finite ending in a terminal state, or infinite
  - a path fragment is initial if $s_0 \in I$
  - $Paths(s)$ is the set of maximal path fragments π with $first(\pi) = s$

Traces

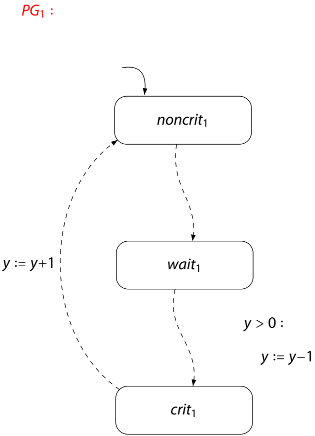
- Actions are mainly used to model the (possibility of) interaction
  - synchronous or asynchronous communication
- Here, focus on the states that are visited during executions
  - the states themselves are not "observable", but just their atomic propositions
- Consider sequences of the form $L(s_0)\,L(s_1)\,L(s_2)\ldots$
  - just register the (set of) atomic propositions that are valid along the execution
  - instead of execution $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$

  $\Rightarrow$ this is called a trace
- For a transition system without terminal states:
  - traces are infinite words over the alphabet 2^{AP}, i.e., they are in $\left(2^{AP}\right)^{\omega}$

Traces

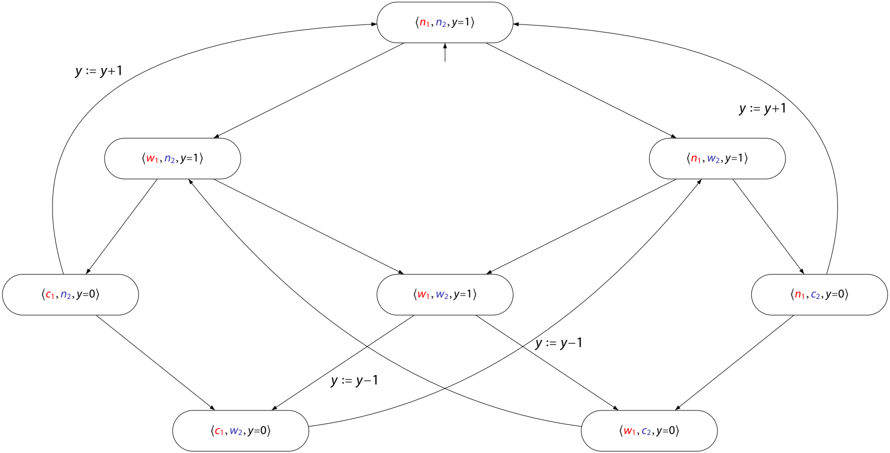
- Let transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states
  - all maximal paths (and excutions) are infinite
- The trace of path fragment $\pi = s_0 \, s_1 \ldots$ is $trace(\pi) = L(s_0) \, L(s_1) \ldots$
  - the trace of $\widehat{\pi} = s_0 \, s_1 \ldots s_n$ is $trace(\widehat{\pi}) = L(s_0) \, L(s_1) \ldots L(s_n)$
- The set of traces of a set Π of paths: $trace(\Pi) = \{ \, trace(\pi) \mid \pi \in \Pi \, \}$
- $Traces(s) = trace(Paths(s))$ $\quad Traces(TS) = \bigcup_{s \in I} Traces(s)$
- $Traces_{fin}(s) = trace(Paths_{fin}(s))$ $\quad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s)$

Semaphore-based mutual exclusion



y=0 means "lock is currently possessed"; y=1 means "lock is free"

Transition system $TS(PG_1 \,|||\, PG_2)$



Example traces

Let $AP = \{ crit_1, crit_2 \}$
Example path:

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

The trace of this path is the infinite word:

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

The 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:

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

Linear-time properties

- Linear-time properties specify the traces that a TS may exhibit
  - LT-property specifies the admissible behaviour of system under consideration

    later, a logic will be introduced for specifying LT properties
- A linear-time property (LT property) over AP is a subset of $\left(2^{AP}\right)^{\omega}$
  - finite words are not needed, as it is assumed that there are no terminal states
- TS (over AP) satisfies LT property P (over AP):

$$TS \vDash P \quad \text{if and only if} \quad Traces(TS) \subseteq P$$

  - TS satisfies the LT property P if all its "observable" behaviors are admissible
  - state $s \in S$ satisfies P, notation $s \vDash P$, whenever $Traces(s) \subseteq P$

How to specify mutual exclusion?

"Always at most one process is in its critical section"

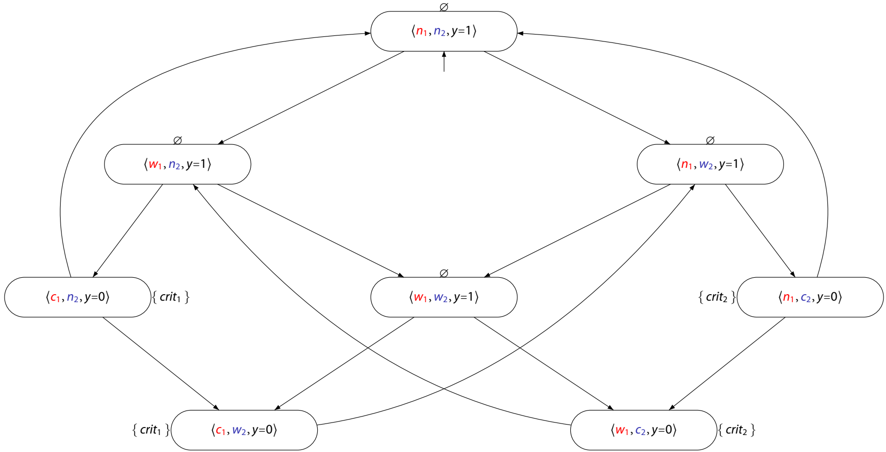
- Let $AP = \{ crit_1, crit_2 \}$
  - other atomic propositions are not of any relevance for this property
- Formalization as LT property

$$P_{mutex} \; = \; \text{set of infinite words } A_0 \, A_1 \, A_2 \ldots \quad \text{with } \{ crit_1, crit_2 \} \not\subseteq A_i \text{ for all } 0 \leq i$$

- Contained in P_{mutex} are e.g., the infinite words:
  - $(\{ crit_1 \} \{ crit_2 \})^\omega$ and $\{ crit_1 \} \{ crit_1 \} \{ crit_1 \} \ldots$ and $\varnothing \varnothing \varnothing \ldots$
  - but not $\{ crit_1 \} \varnothing \{ crit_1, crit_2 \} \ldots$ or $\varnothing \{ crit_1 \}, \varnothing \varnothing \{ crit_1, crit_2 \} \varnothing \ldots$

Does the semaphore-based algorithm satisfy P_{mutex}?

Does the semaphore-based algorithm satisfy P_{mutex}?



Yes as there is no reachable state labeled with $\{ crit_1, crit_2 \}$

How to specify starvation freedom?

"A process that wants to enter the critical section is eventually able to do so'"

- Let $AP = \{ wait_1, crit_1, wait_2, crit_2 \}$
- Formalization as LT-property

$$P_{nostarve} = \text{ set of infinite words } A_0 \, A_1 \, A_2 \ldots \text{ such that:}$$

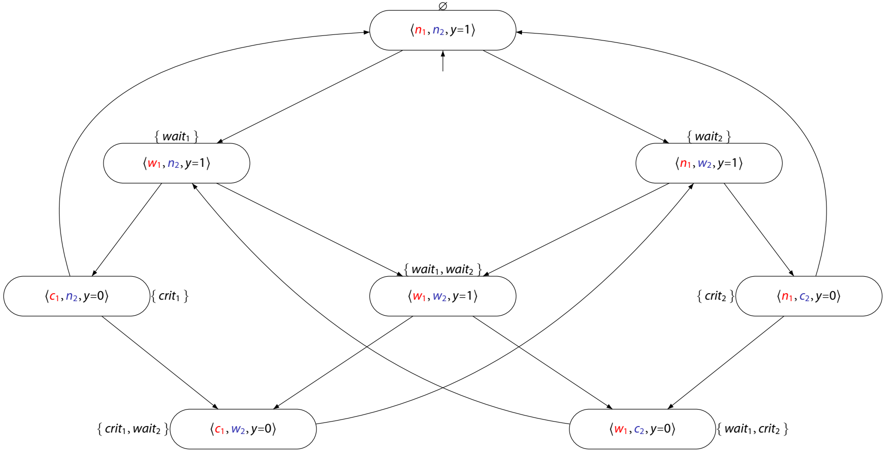
$$\left(\overset{\infty}{\exists} \, j. \, wait_i \in A_j \right) \Rightarrow \left(\overset{\infty}{\exists} \, j. \, crit_i \in A_j \right) \quad \text{for each } i \in \{ 1, 2 \}$$

there exist infinitely many:

$$\left(\overset{\infty}{\exists} \, j. \, wait_i \in A_j \right) \equiv (\forall k \geq 0. \, \exists j > k. \, wait_i \in A_j)$$

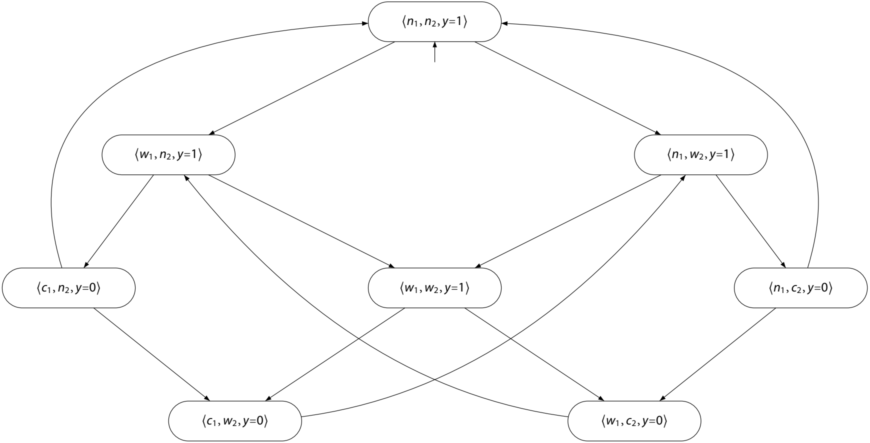
Does the semaphore-based algorithm satisfy $P_{nostarve}$?

Does the semaphore-based algorithm satisfy $P_{nostarve}$?



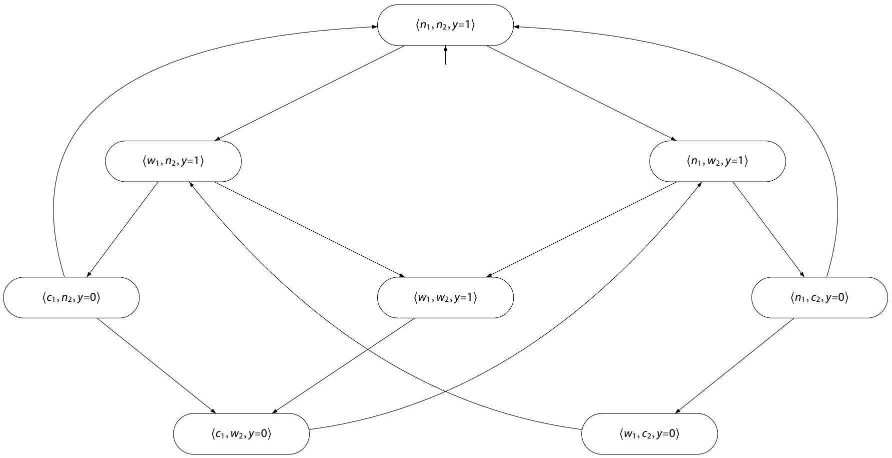
No. Trace $\varnothing\ (\{ wait_2 \}\ \{ wait_1, wait_2 \}\ \{ crit_1, wait_2 \})^\omega \in Traces(TS)$, but $\notin P_{nostarve}$

Mutual exclusion algorithm revisited



this algorithm satisfies P_{mutex}

Refining the mutual exclusion algorithm



this variant algorithm with an omitted edge also satisfies P_{mutex}

Trace equivalence and LT properties

For TS and TS' be transition systems (over AP) without terminal states:

$Traces(TS) \subseteq Traces(TS')$

if and only if

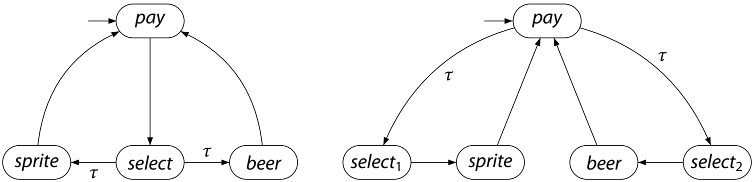
for any LT property P: $TS' \models P$ implies $TS \models P$

$Traces(TS) = Traces(TS')$

if and only if

TS and TS' satisfy the same LT properties

Two beverage vending machines



$$AP = \{ pay, sprite, beer \}$$

there is no LT-property that can distinguish between these machines

Invariants

- Safety properties ≈ "nothing bad should happen" [Lamport 1977]
- Typical safety property: mutual exclusion property
  - the bad thing (having >1 process in the critical section) never occurs
- Another typical safety property is deadlock freedom

⇒ These properties are in fact invariants

- An invariant is an LT property
  - that is given by a condition Φ for the states
  - and requires that Φ holds for all reachable states
  - e.g., for mutex property $\Phi \equiv \neg crit_1 \lor \neg crit_2$

Invariants

- An LT property P_{inv} over AP is an invariant if there is a propositional logic formula Φ over AP such that:

$$P_{inv} = \left\{ A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^{\omega} \;\middle|\; \forall j \geq 0.\, A_j \models \Phi \right\}$$

  - Φ is called an invariant condition of P_{inv}
- Note that

$$\begin{array}{lll} TS \models P_{inv} & \text{iff} & trace(\pi) \in P_{inv} \text{ for all paths } \pi \text{ in } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \text{ that belong to a path of } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \in Reach(TS) \end{array}$$

- Φ has to be fulfilled by all initial states and
  - satisfaction of Φ is invariant under all transitions in the reachable fragment of TS

Checking an invariant

- Checking an invariant for the propositional formula Φ
  - = check the validity of Φ in every reachable state
  - $\Rightarrow$ use a slight modification of standard graph traversal algorithms (DFS and BFS)
  - provided the given transition system *TS* is finite
- Perform a forward depth-first search
  - at least one state s is found with $s \not\models \Phi \Rightarrow$ the invariance of Φ is violated
- Alternative: backward search
  - starts with all states where Φ does not hold
  - calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \not\models \Phi} Pre^*(s)$

A naive invariant checking algorithm

Require: finite transition system *TS* and propositional formula Φ
Ensure: true if *TS* satisfies the invariant "always Φ", otherwise false

---

set of state $R := \varnothing$; {the set of visited states}
stack of state $U := \varepsilon$; {the empty stack}
bool $b :=$ true; {all states in R satisfy Φ}
for all $s \in I$ **do**
    if $s \notin R$ **then**
        visit(s) {perform a dfs for each unvisited initial state}
    end if
end for
return b

A naive invariant checking algorithm

process visit (state s)
 $push(s, U)$; {push s on the stack}
 $R := R \cup \{ s \}$; {mark s as reachable}
 repeat
  $s' := top(U)$;
  $b := b \wedge (s' \models \Phi)$; {check validity of Φ in s'}
  if $Post(s') \subseteq R$ **then**
   $pop(U)$;
  else
   let $s'' \in Post(s') \setminus R$
   $push(s'', U)$;
   $R := R \cup \{ s'' \}$; {state s'' is a new reachable state}
  end if
 until $(U = \varepsilon)$ **endproc**

error indication is state refuting Φ

initial path fragment $s_0\, s_1\, s_2 \ldots s_n$ with $s_i \models \Phi$ $(i \neq n)$ and $s_n \not\models \Phi$ is more useful

Invariant checking by DFS

Require: finite transition system *TS* and propositional formula Φ
Ensure: "yes" if $TS \models$ "always Φ", otherwise "no" plus a counterexample

---

set of states $R := \varnothing$; {the set of reachable states}
stack of states $U := \varepsilon$; {the empty stack}
bool $b :=$ true; {all states in R satisfy Φ}
while $(I \setminus R \neq \varnothing \wedge b)$ **do**
  let $s \in I \setminus R$; {choose an arbitrary initial state not in R}
  visit(s); {perform a DFS for each unvisited initial state}
end while
if b **then**
  return("yes") {$TS \models$ "always Φ"}
else
  return("no", reverse(U)) {counterexample arises from the stack content}
end if

Invariant checking by DFS

process visit (state s)
$push(s, U)$; {push s on the stack}
$R := R \cup \{ s \}$; {mark s as reachable}
repeat
    $s' := top(U)$;
    $b := b \wedge (s' \vDash \Phi)$; {check validity of Φ in s'}
    if $Post(s') \subseteq R$ **then**
        $pop(U)$;
    else
        let $s'' \in Post(s') \setminus R$
        $push(s'', U)$;
        $R := R \cup \{ s'' \}$; {state s'' is a new reachable state}
    end if
until $((U = \varepsilon) \vee \neg b)$ **endproc**

Time complexity

- Under the assumption that
  - $s' \in Post(s)$ can be encountered in time $\Theta(|Post(s)|)$

  $\Rightarrow$ this holds for a representation of $Post(s)$ by adjacency lists
- The time complexity for invariant checking is $\mathcal{O}(\, N * (1 + |\Phi|) + M \,)$
  - where N denotes the number of reachable states, and
  - $M = \sum_{s \in S} |Post(s)|$ the number of transitions in the reachable fragment of TS
- The adjacency lists are typically given implicitly
  - e.g., by a syntactic description of the concurrent processes as program graphs
  - $Post(s)$ is obtained by the rules for the transition relation

Safety properties

- Safety properties may impose requirements on finite path fragments
  - and cannot be verified by considering the reachable states only
- A safety property which is not an invariant:
  - consider a cash dispenser, also known as automated teller machine (ATM)
  - property "money can only be withdrawn once a correct PIN has been provided"

  $\Rightarrow$ not an invariant, since it is not a state property
- But a safety property:
  - any infinite run violating the property has a finite prefix that is "bad"
  - i.e., in which money is withdrawn without issuing a PIN before

Safety properties

- LT property P_{safe} over AP is a safety property if
  - for all $\sigma \in \left(2^{AP}\right)^{\omega} \setminus P_{safe}$ there exists a finite prefix $\widehat{\sigma}$ of σ such that:

$$P_{safe} \cap \underbrace{\left\{\sigma' \in \left(2^{AP}\right)^{\omega} \mid \widehat{\sigma} \text{ is a prefix of } \sigma'\right\}}_{\text{all possible extensions of } \widehat{\sigma}} = \varnothing$$

  - any such finite word $\widehat{\sigma}$ is called a bad prefix for P_{safe}
- Minimal bad prefix for P_{safe}:
  - is a bad prefix $\widehat{\sigma}$ for P_{safe} for which no proper prefix of $\widehat{\sigma}$ is a bad prefix for P_{safe}

  $\Rightarrow$ minimal bad prefixes are bad prefixes of minimal length

Safety properties and finite traces

For transition system TS without terminal states

and safety property P_{safe}:

$$TS \models P_{safe} \text{ if and only if } Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing$$

where $BadPref(P_{safe})$ is the set of bad prefixes of P_{safe}

Closure

- For trace $\sigma \in \left(2^{AP}\right)^{\omega}$, let $pref(\sigma)$ be the set of finite prefixes of σ:

$$pref(\sigma) = \{\, \widehat{\sigma} \in \left(2^{AP}\right)^{*} \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma \,\}$$

  - if $\sigma = A_0 A_1 \ldots$ then $pref(\sigma) = \left\{\varepsilon, A_0, A_0 A_1, A_0 A_1 A_2, \ldots\right\}$ is infinite
- For property P this is lifted as follows: $pref(P) = \bigcup_{\sigma \in P} pref(\sigma)$
- The closure of LT property P:

$$closure(P) \;=\; \left\{\sigma \in \left(2^{AP}\right)^{\omega} \mid pref(\sigma) \subseteq pref(P)\right\}$$

  - the set of infinite traces whose finite prefixes are also prefixes of P, or
  - infinite traces in the closure of P do not have a prefix that is not a prefix of P

Safety properties and closures

LT property P over AP is a safety property

if and only if $closure(P) = P$

Finite trace equivalence and safety properties

For TS and TS' be transition systems (over AP) without terminal states:

$$Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$$

if and only if

for any safety property P_{safe} : $TS' \models P_{safe} \;\Rightarrow\; TS \models P_{safe}$

$$Traces_{fin}(TS) = Traces_{fin}(TS')$$

if and only if

TS and TS' satisfy the same safety properties

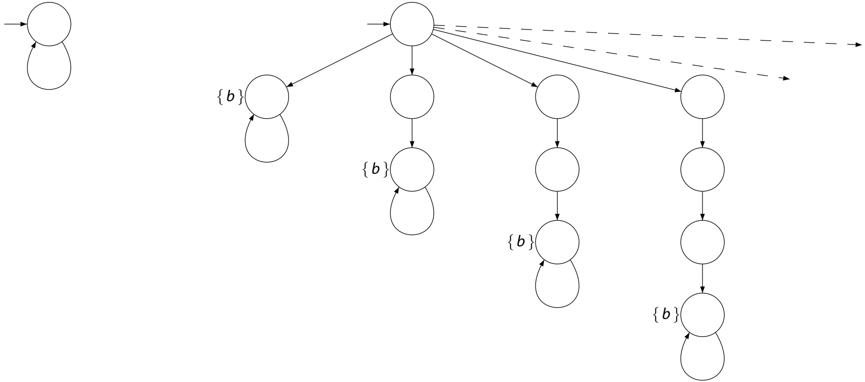
Finite vs. infinite traces

For TS without terminal states and finite TS'

trace inclusion and finite-trace inclusion coincide

this does not hold for infinite TS' (cf. next slide)
but also holds for image-finite TS'

Trace inclusion ≠ finite trace inclusion



$$Traces(TS) \not\subseteq Traces(TS') \quad \text{and} \quad Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$$

Why liveness?

- Safety properties specify that “something bad never happens”
- Doing nothing easily fulfills a safety property
  - as this will never lead to a “bad” situation

$\Rightarrow$ Safety properties are complemented by liveness properties
  - that require some progress
- Liveness properties assert that:
  - ”something good” will happen eventually [Lamport 1977]

The meaning of liveness

[Lamport 2000]

The question of whether a real system satisfies a liveness property is meaningless; it can be answered only by observing the system for an infinite length of time, and real systems don't run forever.

Liveness is always an approximation to the property we really care about. We want a program to terminate within 100 years, but proving that it does would require addition of distracting timing assumptions.

So, we prove the weaker condition that the program eventually terminates. This doesn't prove that the program will terminate within our lifetimes, but it does demonstrate the absence of infinite loops.

Liveness properties

LT property P_{live} over AP is a liveness property whenever

$$pref(P_{live}) = \left(2^{AP}\right)^*$$

- A liveness property is an LT property
  - that does not rule out any prefix
- Liveness properties are violated in "infinite time"
  - whereas safety properties are violated in finite time
  - finite traces are of no use to decide whether P holds or not
  - any finite prefix can be extended such that the resulting infinite trace satisfies P