Linear-Time Properties

Hao Zheng

Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
Email: zheng@cse.usf.edu
Phone: (813)974-4757
Fax: (813)974-5456
We now consider the properties.
Overview

1. Deadlock (Section 3.1)

2. Linear Time Behavior (Section 3.2)
   - Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
   - Linear-time Properties (Section 3.2.3 - 3.2.4)

3. Safety and Invariants (Section 3.3.1 - 3.3.2)

4. Liveness Properties (Section 3.4.1)

5. Fairness (Section 3.5.1)
Deadlock (Section 3.1)

Linear Time Behavior (Section 3.2)
- Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
- Linear-time Properties (Section 3.2.3 - 3.2.4)

Safety and Invariants (Section 3.3.1 - 3.3.2)

Liveness Properties (Section 3.4.1)

Fairness (Section 3.5.1)
Deadlock

- Sequential programs without infinite loops terminate.
- For reactive systems, terminal states are undesirable and represent an error.
  - Embedded controllers need to operate without interruption for a long time.
- A **deadlock** occurs if a system stops while at least one component is in a (local) nonterminal state.
  - System has halted when at least one component should continue.
- Typically occurs when components mutually wait for each other.
Example Deadlock Situation

Consider the parallel composition of two transition systems $TrLight_1 \parallel TrLight_2$ modeling the traffic light intersections at roads. Both traffic lights synchronize by means of the actions $\alpha$ and $\beta$ that indicate the change of light (see Figure 3.1). The apparently trivial error to let both traffic lights turn red results in a deadlock.

While the first traffic light is waiting for synchronization with action $\alpha$, the second traffic light is blocked, since it is waiting to be synchronized with action $\beta$.

Example 3.2. Dining Philosophers

This example, originated by Dijkstra, is one of the most prominent examples in the field of concurrent systems.
Five philosophers are sitting at a round table with a bowl of rice in the middle. For the philosophers (being a little unworldly) life consists of thinking and eating (and waiting, as we will see). To take some rice out of the bowl, a philosopher needs two chopsticks. In between two neighboring philosophers, however, there is only a single chopstick. Thus, at any time only one of two neighboring philosophers can eat. Of course, the use of the chopsticks is exclusive and eating with hands is forbidden.

Note that a deadlock scenario occurs when all philosophers possess a single chopstick. The problem is to design a protocol for the philosophers, such that the complete system is deadlock-free, i.e., at least one philosopher can eat and think infinitely often. Additionally, a solution may require that each philosopher be able to think infinitely often. The latter characteristic is called freedom of individual starvation.

The following obvious design cannot ensure deadlock freedom. Assume the philosophers and the chopsticks are numbered from 0 to 4. Furthermore, assume all following calculations be "modulo 5", e.g., chopstick \( i - 1 \) for \( i = 0 \) denotes chopstick 4, and so on.

Philosopher \( i \) has stick \( i \) on his left and stick \( i - 1 \) on his right. The action request \( i, i \) expresses that stick \( i \) is picked up by philosopher \( i \). Accordingly, request \( i - 1, i \) denotes the action by means of which philosopher \( i \) picks up the \( (i - 1) \)th stick. The actions release \( i, i \) and release \( i - 1, i \) have a corresponding meaning.

The behavior of philosopher \( i \) (called process \( \text{Phil} \_i \)) is specified by the transitions system depicted in the left part of Figure 3.2. Solid arrows depict the synchronizations with the \( i \)th stick, dashed arrows refer to communications with the \( i - 1 \)th stick. The sticks are modeled as independent processes (called \( \text{Stick} \_i \)) with which the philosophers synchronize via actions request and release; see either part of Figure 3.2 that represent the process of stick \( i \). A stick process prevents philosopher \( i \) from picking up the \( i \)th stick when philosopher \( i + 1 \) is using it.

- Design a protocol which is deadlock-free.
- Design a protocol which is free of starvation.
A Transition System for the Dining Philosophers

Figure 3.2: Transition systems for the $i$th philosopher and the $i$th stick.
The complete system is of the form:

\[
\text{Phil}_4 \parallel \text{Stick}_3 \parallel \text{Phil}_3 \parallel \text{Stick}_2 \parallel \text{Phil}_2 \parallel \text{Stick}_1 \parallel \text{Phil}_1 \parallel \text{Stick}_0 \parallel \text{Phil}_0 \parallel \text{Stick}_4
\]

This (initially obvious) design leads to a deadlock situation, e.g., if all philosophers pick up their left stick at the same time. A corresponding execution leads from the initial state \(\langle \text{think}_4, \text{avail}_3, \text{think}_3, \text{avail}_2, \text{think}_2, \text{avail}_1, \text{think}_1, \text{avail}_0, \text{think}_0, \text{avail}_4 \rangle\) by means of the action sequence \(\text{request}_4, \text{request}_3, \text{request}_2, \text{request}_1, \text{request}_0\) (or any other permutation of these 5 request actions) to the terminal state \(\langle \text{wait}_4, \text{0, occupied}_4, \text{wait}_3, \text{occupied}_4, \text{wait}_2, \text{occupied}_3, \text{wait}_1, \text{occupied}_2, \text{wait}_0, \text{occupied}_1 \rangle\).

This terminal state represents a deadlock with each philosopher waiting for the needed stick to be released.

A possible solution to this problem is to make each stick available for only one philosopher at a time. The corresponding chopstick process is depicted in the right part of Figure 3.3.

In state \(\text{available}_{i,j}\) only philosopher \(j\) is allowed to pick up the \(i\)th stick. The above-mentioned deadlock situation can be avoided by the fact that some sticks (e.g., the first, the third, and the fifth stick) start in state \(\text{available}_{i,i}\), while the remaining sticks start in state \(\text{available}_{i,i+1}\). It can be verified that this solution is deadlock- and starvation-free.

\[req_{4,4}, req_{3,3}, req_{2,2}, req_{1,1}, req_{0,0}\] leads to a deadlock.
Figure 3.3: Improved variant of the \( i \)th philosopher and the \( i \)th stick.

- Even and odd numbered sticks start in different available states.
Contents

1. Deadlock (Section 3.1)

2. Linear Time Behavior (Section 3.2)
   - Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
   - Linear-time Properties (Section 3.2.3 - 3.2.4)

3. Safety and Invariants (Section 3.3.1 - 3.3.2)

4. Liveness Properties (Section 3.4.1)

5. Fairness (Section 3.5.1)
Recall Executions

- A **finite execution fragment** $\varrho$ of $TS$ is an alternating sequence of states and actions ending with a state:

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

- An **infinite execution fragment** $\rho$ of $TS$ is an infinite, alternating sequence of states and actions:

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

- An **execution** of $TS$ is an **initial**, **maximal** execution fragment
  - An execution fragment is **initial** if $s_0 \in I$.
  - A maximal execution fragment can be finite, ending in a terminal state, or infinite.
Traces

- Let transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states (i.e., all executions are infinite).
  - Terminal states are assumed to have self-loop transitions.
- The $trace$ of execution $\rho = s_0 \alpha_0 s_1 \alpha_1 \ldots$ is
  $$trace(\pi) = L(s_0) L(s_1) \ldots.$$
- The trace of $s_0 \alpha_0 s_1 \alpha_1 \ldots s_n$ is
  $$trace(\hat{\pi}) = L(s_0) L(s_1) \ldots L(s_n).$$
- Traces of a $TS$ are infinite words over the alphabet $2^{AP}$, i.e.,
  $$Traces(TS) \subseteq (2^{AP})^\omega.$$
Semaphore-Based Mutual Exclusion

\[ PG_1 : \]

\[ y := y + 1 \]
\[ noncrit_1 \]
\[ wait_1 \]
\[ crit_1 \]
\[ y > 0 : \]
\[ y := y - 1 \]

\[ PG_2 : \]

\[ y := y + 1 \]
\[ noncrit_2 \]
\[ wait_2 \]
\[ crit_2 \]
\[ y > 0 : \]
\[ y := y - 1 \]

\( y = 0 \) means “lock is currently possessed”; \( y = 1 \) means “lock is free”
\(TS(PG_1 \parallel\| PG_2)\)
Let \( AP = \{ \text{crit}_1, \text{crit}_2 \} \)

The trace of the finite execution:

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

is: \( \text{trace}(\hat{\pi}) = \emptyset \emptyset \emptyset \{ \text{crit}_2 \} \emptyset \{ \text{crit}_1 \} \)
Let $AP = \{ \texttt{crit}_1, \texttt{crit}_2 \}$

The trace of the infinite execution:

$$\pi = \langle n_1, n_2, y = 1 \rangle \rightarrow \langle w_1, n_2, y = 1 \rangle \rightarrow \langle c_1, n_2, y = 0 \rangle \rightarrow \langle n_1, n_2, y = 1 \rangle \rightarrow \langle n_1, w_2, y = 1 \rangle \rightarrow \langle n_1, c_2, y = 0 \rangle \rightarrow \ldots$$

is: $\text{trace}(\pi) = \emptyset \emptyset \{ \texttt{crit}_1 \} \emptyset \emptyset \{ \texttt{crit}_2 \} \emptyset \emptyset \{ \texttt{crit}_1 \} \emptyset \emptyset \{ \texttt{crit}_2 \} \ldots$
Linear-Time Properties

- Linear-time properties specify the traces that a TS should only exhibit.

  A \textit{linear-time property} (LT property) $P$ over $AP$ is a subset of $(2^{AP})^\omega$.

  - Finite words are not needed assuming there are \textit{no terminal states}.
  - A trace satisfies LT property $P$ if it is included in $P$.

\[ TS \ (\text{over } AP) \ \textit{satisfies} \ \text{LT property} \ P \ (\text{over } AP): \]
\[ TS \models P \ \text{if and only if} \ \ Traces(TS) \subseteq P \]

- $TS$ satisfies the LT property $P$ if all its traces are admissible.

- Later, a logic will be introduced for specifying LT properties.
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 relevant for this property.
- Formalization as LT property:
  $$P_{mutex} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{ with } \{ crit_1, crit_2 \} \not\subseteq A_i \text{ for all } 0 \leq i$$

Which of the following infinite words satisfies $P_{mutex}$?

- $(\{ crit_1 \} \{ crit_2 \})^\omega$
- $\{ crit_1 \} \{ crit_1 \} \{ crit_1 \} \ldots$
- $\emptyset \emptyset \emptyset \ldots$
- $\{ crit_1 \} \emptyset \{ crit_1, crit_2 \} \ldots$
- $\emptyset \{ crit_1 \} \emptyset \emptyset \{ crit_1, crit_2 \} \emptyset \ldots$
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 relevant for this property.
- Formalization as LT property:
  $$P_{mutex} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{ with } \{ crit_1, crit_2 \} \not\subseteq A_i \text{ for all } 0 \leq i$$

Which of the following infinite words satisfies $P_{mutex}$?

- $(\{ crit_1 \} \{ crit_2 \})^\omega$ ✓
- $\{ crit_1 \} \{ crit_1 \} \{ crit_1 \} \ldots$ ✓
- $\emptyset \emptyset \emptyset \ldots$ ✓
- $\{ crit_1 \} \emptyset \{ crit_1, crit_2 \} \ldots$ ✗
- $\emptyset \{ crit_1 \} \emptyset \emptyset \{ crit_1, crit_2 \} \emptyset \ldots$ ✗
Does the Semaphore-Based Algorithm Satisfy $P_{mutex}$?

\[ TS (PG_{1} ||| PG_{2}) \]

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

\[ y := y+1 \]
\[ y := y-1 \]
\[ y := y+1 \]

\[ y := y-1 \]
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 $AP = \{ \text{wait}_1, \text{crit}_1, \text{wait}_2, \text{crit}_2 \}$
- Formalization #1:

  \[
  P_{\text{finwait}} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{such that:}
  \]

  \[
  \forall j. (\text{wait}_i \in A_j \Rightarrow \exists k \geq j. \text{crit}_i \in A_k ) \quad \text{for each } i \in \{1, 2\}
  \]

- However, it does not specify that a process should wait often.
- This property holds if process $i$ never wants to enter the critical section!
Does the Semaphore-Based Algorithm Satisfy $P_{nostarve}$?

\[ \langle n_1, n_2, y=1 \rangle \]

\[ \langle w_1, n_2, y=1 \rangle \]

\[ \langle n_1, w_2, y=1 \rangle \]

\[ \langle c_1, n_2, y=0 \rangle \]

\[ \langle w_1, w_2, y=1 \rangle \]

\[ \langle n_1, c_2, y=0 \rangle \]

\[ \langle c_1, w_2, y=0 \rangle \]

\[ \langle w_1, c_2, y=0 \rangle \]

\[ y := y + 1 \]

\[ y := y - 1 \]
Does the Semaphore-Based Algorithm Satisfy $P_{nostarve}$?

No. $\emptyset \ (\{wait_2\} \{wait_1, wait_2\} \{crit_1, wait_2\})^\omega \in \text{Traces}(TS)$, but $\not\in P_{nostarve}$. 
Theorem 3.15

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

$$\text{Traces}(TS) \subseteq \text{Traces}(TS')$$

if and only if

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

- $\text{Traces}(TS) \subseteq \text{Traces}(TS')$ means that $TS$ is an implementation of $TS'$
- $TS$ is also referred to as refinement of $TS'$. 
Mutual Exclusion Algorithm Revisited $T S'$

- $y := y + 1$
- $y := y + 1$
- $y := y - 1$
- $y := y - 1$

Diagram: States and transitions illustrating the mutual exclusion algorithm.
This algorithm satisfies $P_{mutex}$ as $Traces(TS) \subset Traces(TS')$ and $TS' \models P_{mutex}$. 
Corollary 3.18 Trace Equivalence and LT Properties

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

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

if and only if

$TS$ and $TS'$ satisfy the same LT properties

- $TS$ and $TS'$ cannot be distinguished by any LT properties.
Two Beverage Vending Machines

There is no LT-property that can distinguish between these machines.

\[ AP = \{ \text{pay, soda, beer} \} \]

There is no LT-property that can distinguish between these machines.
1. Deadlock (Section 3.1)

2. Linear Time Behavior (Section 3.2)
   - Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
   - Linear-time Properties (Section 3.2.3 - 3.2.4)

3. Safety and Invariants (Section 3.3.1 - 3.3.2)

4. Liveness Properties (Section 3.4.1)

5. Fairness (Section 3.5.1)
Safety properties \( \approx \) “nothing bad should happen”. [Lamport 1977]

Typical safety property: mutual exclusion property.
- The bad thing (having \( \geq 1 \) 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 \( \Phi \) for the states and requires that \( \Phi \) holds for all reachable states (e.g., for mutex property \( \Phi \equiv \neg crit_1 \lor \neg crit_2 \)).

Read section 3.3.1. Skim over 3.3.2. Ignore 3.3.3.
3.3.1 Invariants

Definition 3.20  Invariant

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

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

where $\Phi$ is called an invariant condition of $P_{inv}$. 
Example Invariants

- Mutual exclusion
  \[ \Phi = \neg crit_1 \lor \neg crit_2 \]

- Deadlock freedom in Dining Philosophers
  \[ \Phi = \neg wait_0 \lor \ldots \lor \neg wait_4 \]

Deadlock is avoided if at least one philosopher is not waiting to pick up sticks.
Notes on Invariants

\[ TS \models P_{inv} \iff \text{trace}(\pi) \in P_{inv} \text{ for all paths } \pi \text{ in } TS \]
\[ \text{iff } L(s) \models \Phi \text{ for all states } s \text{ that belong to a path of } TS \]
\[ \text{iff } L(s) \models \Phi \text{ for all states } s \in \text{Reach}(TS) \]

- \( \Phi \) has to be fulfilled by all initial states and satisfaction of \( \Phi \) is invariant under all transitions in the reachable fragment of \( TS \).
• Checking an invariant for the propositional formula $\Phi$
  $\implies$ Check the validity of $\Phi$ in every reachable state.
  $\implies$ Use a slight modification of standard graph traversal algorithms (i.e.,
  \textit{depth-first search} (DFS) and \textit{breadth-first search} (BFS)).
  • Provided that the given transition system $TS$ is finite.

• Perform a forward depth-first search:
  • If any state $s$ is found with $s \not\models \Phi \implies$ the invariance of $\Phi$ is violated.

• Alternative is to perform a backward search:
  • Starts with all states where $\Phi$ does not hold.
  • Calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \not\models \Phi} \text{Pre}^*(s)$.
  • If there is a $\text{init} \in I$ such that $\text{init} \in \bigcup_{s \in S, s \not\models \Phi} \text{Pre}^*(s)$, then $\Phi$ is violated.
Algorithm 3: A Naive Invariant Checking Algorithm

\[ R = \emptyset; \]  // Set of reachable states
\[ U = \emptyset; \]  // Stack of states
\[ \text{bool } b := \text{true}; \]  // All states in \( R \) satisfies \( \Phi \)

\textbf{foreach } s \in I \textbf{ do}

\hspace{1em} \textbf{if } s \not\in R \textbf{ then}

\hspace{2em} \text{visit}(s);  // \text{visit}(s) shown on the next slide

\text{return } b;
The \textit{visit}(s) Procedure

\begin{verbatim}
push(s, U); // Set of reachable states
R := R ∪ \{s\}; // Stack of states
bool b := true; // All states in R satisfies Φ
while U ≠ ∅ do
  s′ = top(U);
  if Post(s′) ⊆ R then
    pop(U);
    b := b ∧ (s′ |= Φ);
  else
    Let s'' ∈ Post(s′)\R;
    push(s'', U);
    R := R ∪ \{s''\};
return b;
\end{verbatim}

\textit{Error indication is state that refutes Φ.}
\textit{s_0 s_1 \ldots s_n with s_i |= Φ (i ≠ n) and s_n \not|= Φ is a counter-example.}
$\langle n_1, n_2, y = 1 \rangle$

$\langle w_1, n_2, y = 1 \rangle$

$\langle n_1, w_2, y = 1 \rangle$

$\langle c_1, n_2, y = 0 \rangle$

$\langle w_1, w_2, y = 1 \rangle$

$\langle n_1, c_2, y = 0 \rangle$

$\langle c_1, w_2, y = 0 \rangle$

$\langle w_1, c_2, y = 0 \rangle$

$y := y + 1$

$y := y - 1$

$y := y + 1$
The time complexity for invariant checking is:

$$O(N \ast (1 + |\Phi|) + M)$$

where

- $N$ is the number of reachable states,
- $M$ is the number of transitions in the reachable fragment of $TS$, and
- $|\Phi|$ is the length of $\Phi$. 
3.3.2 Safety Properties

- Safety properties can be translated to invariants.
  - Safety properties need to hold in every state!
- A TS violates a LT safety property $P_{safe}$ if there is a BadPref such that

$$\text{BadPref} \cdot (2^{AP})^\omega \notin P_{safe}.$$ 

where BadPref is a finite trace that ends with a violation to $P_{safe}$.
- The idea is different from what is given in the book.

$P_{safe}$: red preceded immediately by yellow.

Counter-example for $P_{safe}$: •••
Contents

1. Deadlock (Section 3.1)

2. Linear Time Behavior (Section 3.2)
   - Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
   - Linear-time Properties (Section 3.2.3 - 3.2.4)

3. Safety and Invariants (Section 3.3.1 - 3.3.2)

4. Liveness Properties (Section 3.4.1)

5. Fairness (Section 3.5.1)
Liveness Properties

- Safety properties specify that “something bad never happens”.
- Doing nothing easily fulfills a safety property as this will never lead to a “bad” situation.
- Safety properties are complemented by liveness properties that require some progress.
  - Safety violations are characterized by finite traces.
- Liveness properties assert that ”something good will happen”.
  - Liveness violations are characterized by infinite traces.
Example Liveness Properties

- “If the tank is empty, the outlet valve will eventually be closed”.
- “If the outlet valve is open and the request signal disappears, the outlet valve will eventually be closed”.
- “If the tank is full and a request is present, the outlet valve will eventually be opened”.
- “The program terminates within 31 computational steps”.
  \[\Rightarrow\] A finite trace may violate this; this is a safety property!
- “The program eventually terminates”.
Liveness Properties for Mutual Exclusion

- **Eventually:**
  - Each process will eventually enter its critical section.

- **Repeated eventually:**
  - Each process will enter its critical section infinitely often.

- **Starvation freedom:**
  - Each waiting process will eventually enter its critical section.

*How to formalize these properties?*
$P = \{ A_0, A_1, A_2 \ldots | A_j \subseteq AP \& \ldots \}$ and $AP = \{ wait_1, crit_1, wait_2, crit_2 \}$

- **Eventually:**
  
  _Each process will eventually enter its critical section._

  $$(\exists j \geq 0. \ crit_1 \in A_j) \land (\exists j \geq 0. \ crit_2 \in A_j)$$

- **Repeated eventually:**
  
  _Each process will enter its critical section infinitely often._

  $$\left(\bar{\exists} j \geq 0. \ crit_1 \in A_j\right) \land \left(\bar{\exists} j \geq 0. \ crit_2 \in A_j\right)$$

- **Starvation freedom:**
  
  _Each waiting process will eventually enter its critical section._

  $$\forall j \geq 0. \ (wait_1 \in A_j \Rightarrow (\exists k > j. crit_1 \in A_k)) \land \forall j \geq 0. \ (wait_2 \in A_j \Rightarrow (\exists k > j. crit_2 \in A_k))$$
Summary LT Properties

- LT properties are finite sets of infinite words over $2^{\mathcal{AP}}$ (= traces).
- An invariant requires a condition $\Phi$ to hold in any reachable state.
- Each trace refuting a safety property has a finite prefix causing this.
  - Invariants are safety properties with bad prefix $\Phi^* (\neg \Phi)$.
  - A safety property is regular iff its set of bad prefixes is a regular language.
  $\Rightarrow$ Safety properties constrain finite behaviors.
- A liveness property does not rule out finite behavior, liveness properties constrain infinite behaviors.
Deadlock (Section 3.1)

Linear Time Behavior (Section 3.2)
- Executions, Paths, and Traces (Section 3.2.1 - 3.2.2)
- Linear-time Properties (Section 3.2.3 - 3.2.4)

Safety and Invariants (Section 3.3.1 - 3.3.2)

Liveness Properties (Section 3.4.1)

Fairness (Section 3.5.1)
Does this Program Always Terminate?

```
Inc ||| Reset

where

proc Inc = while x ≥ 0 do x := x + 1 od
proc Reset = x := -1
```

$x$ is a shared integer variable that initially has value 0
Is it Possible to Starve?

Can either process enter its critical section if it wants to?
• Starvation freedom is often considered under process fairness.
  ⇒ There is a fair scheduling of the execution of processes.

• Fairness is typically needed to prove liveness.
  • Not needed for safety properties!
  • To prove liveness or some form of progress, progress needs to be possible.

• Fairness is concerned with a fair resolution of nondeterminism such that it is not biased to consistently ignore a possible option.

• Problem: liveness properties constrain infinite behaviors, but some traces—that are unfair—refute the liveness property.
3.5.1 Fairness Constraints

- **Unconditional fairness**
  An activity is executed infinitely often.

- **Strong fairness (compassion)**
  If an activity is *infinitely often* enabled (not necessarily always!) then it has to be executed infinitely often.

- **Weak fairness (justice)**
  If an activity is *continuously enabled* (no temporary disabling!) then it has to be executed infinitely often.

We will use actions to distinguish fair and unfair behaviors.
A state-based notion of fairness could also be defined.
Fairness Definition

For \( TS = (S, \text{Act}, \to, I, AP, L) \) without terminal states, \( A \subseteq \text{Act} \),
and infinite execution fragment \( \rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \) of \( TS \):

1. \( \rho \) is unconditionally \( A \)-fair whenever:

\[
\text{true} \implies \forall k \geq 0. \exists j \geq k. \alpha_j \in A
\]

ininitely often \( A \) is taken

2. \( \rho \) is strongly \( A \)-fair whenever:

\[
(\forall k \geq 0. \exists j \geq k. \text{Act}(s_j) \cap A \neq \emptyset) \implies \forall k \geq 0. \exists j \geq k. \alpha_j \in A
\]

ininitely often \( A \) is enabled

3. \( \rho \) is weakly \( A \)-fair whenever:

\[
(\exists k \geq 0. \forall j \geq k. \text{Act}(s_j) \cap A \neq \emptyset) \implies \forall k \geq 0. \exists j \geq k. \alpha_j \in A
\]

\( A \) is eventually always enabled

\[
\text{infinitely often } A \text{ is taken}
\]

where \( \text{Act}(s) = \{ \alpha \in \text{Act} \mid \exists s' \in S. s \xrightarrow{\alpha} s' \} \).
This program terminates under unconditional (process) fairness on process IDs:

\[ \text{proc Inc} = \text{while } x \geq 0 \text{ do } x := x + 1 \text{ od} \]
\[ \text{proc Reset} = x := -1 \]

\(x\) is a shared integer variable that initially has value 0
Another Fairness Example

Does the following property holds?

\[ x \text{ eventually becomes negative.} \]

\[
\begin{align*}
\text{proc Inc} &= \text{while } x \geq 0 \text{ do } \alpha_1 : x := x + 1 \text{ od} \\
\text{proc Reset} &= \text{while } x < 0 \text{ do } \alpha_2 : x := -1 \text{ od}
\end{align*}
\]

\(x\) is a shared integer variable that initially has value 0

1. Using unconditional fairness on \(\alpha_1\) and \(\alpha_2\).
2. Using strong fairness on \(\alpha_1\) and \(\alpha_2\).
Which Fairness Notion to Use?

\[ \text{unconditional } A\text{-fairness} \implies \text{strong } A\text{-fairness} \implies \text{weak } A\text{-fairness} \]

- Fairness constraints aim to rule out “unreasonable” executions.
- Too strong? \(\Rightarrow\) relevant computations ruled out, verification yields:
  - “false”: error found.
  - “true”: don’t know as some relevant execution may refute it.
- Too weak? \(\Rightarrow\) too many computations considered, verification yields:
  - “true”: property holds.
  - “false”: don’t know, as refutation may be due to some unreasonable run.
Example (Un)fair Executions

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

\[
y := y+1
\]

\[
y := y-1
\]
Fairness for Mutual Exclusion

\[ y := y - 1 \]

\[ y := y + 1 \]

\[ \mathcal{F} = (\{\text{enter}_1, \text{enter}_2\}, \emptyset, \emptyset) \]

\[ \mathcal{F}_{\text{unconditional}} \]
Fairness for Mutual Exclusion

\[ F = (\emptyset, \{\text{enter}_1, \text{enter}_2\}, \emptyset) \]

\( F_{\text{strong}} \)