Chapter 3:
Linear-Time Properties

Prof. Ali Movaghar
Verification of Reactive Systems
Spring 91
To verify the transition system model of the system under consideration, we need to specify the property of interest.

We introduce how the linear-time properties can be specified and then model-checked in an automated manner.
Deadlock

- **Sequential** programs that are not subject to divergence (i.e., endless loops) have a *terminal* state.
  - A terminal state is a state without any outgoing transition.
Deadlock(Con.)

- In contrast for parallel systems, computations typically do not terminate.
  - In these systems, terminal states are undesirable and mostly represent a design error.
  - Or these terminal states may indicate a deadlock!
A deadlock occurs if the complete system is in a terminal state, although at least one component is in a (local) non-terminal state.

In other words, the entire system has come to a halt, whereas at least one component has the possibility to continue.

A typical deadlock scenario occurs when components mutually waits for each other to progress.
Deadlock (Con.)

- **Example 3.1**: consider the parallel composition of two transition systems
  \[ \text{TrLight}_1 \parallel \text{TrLight}_2 \]
  modeling the traffic lights of two intersecting roads:
Traffic lights are modeled by following transition systems:

They are synchronized by means of the actions $\alpha$ and $\beta$ that indicate the change of light.
Deadlock(Con.)

- One trivial error is to let both traffic lights start with a red light which results in a deadlock:
  - While the first traffic light is waiting to be synchronized on action $\alpha$, the second light is waiting to be synchronized with action $\beta$. 

\[ TrLight_1 \parallel TrLight_2 \]
\[ \langle \text{red, red} \rangle \]
Deadlock (Con.)

**Example 3.2: Dining philosophers**

- The philosophers life consists of thinking and eating.
- To eat, a philosopher needs two chopsticks.
Deadlock(Con.)

- At any time only one of two neighboring philosophers can eat.
- A deadlock occurs when all philosophers posses a single chopstick.
- The problem is to design a protocol such that the complete system is deadlock-free.
Deadlock(Con.)

- **Deadlock-free** means At least one philosopher can **eat and think infinitely often**.

- Additionally, a **fair solution** may be required with each philosopher being able to think and eat infinitely often.

  - This characteristic is called freedom of **individual starvation**.
Deadlock(Con.)

- The following obvious design cannot ensure deadlock freedom.
- Assume the philosophers and the chopsticks are numbered from 0 to 4.
- Assume the minus operation be “modulo 5”, e.g. i-1 for i=0 denotes 4, and so on.
Deadlock (Con.)

- Philosopher i has stick i on his left and stick i-1 on his right side.
- Action request \( r_{i,i} \) express that stick i is picked up by philosopher i.
- Accordingly request \( r_{i-1,i} \) denotes the action by means of which philosopher i picks up the (i-1)th stick.
- The actions release \( l_{i,i} \) and release \( l_{i-1,i} \) have a corresponding meaning.
The behavior of philosopher \( i \) and sticks are modeled by:

- \( \text{think} \)
- \( \text{wait for left stick} \)
- \( \text{wait for right stick} \)
- \( \text{return the left stick} \)
- \( \text{return the right stick} \)

The process is illustrated in the diagram, showing transitions between states including request, release, available, and occupied.
Deadlock(Con.)

- In these LTSs, the solid arrows depict the synchronizations can occur between the i-th stick (called $\text{Stick}_i$) and philosopher $i$ (called $\text{Phil}_i$), while dashed arrows refer to communications between $\text{phil}_i$ and i-1th stick.

- The complete system is:

$$\text{Phil}_4 || \text{Stick}_3 || \text{Phil}_3 || \text{Stick}_2 || \text{Phil}_2 || \\
\text{Stick}_1 || \text{Phil}_1 || \text{Stick}_0 || \text{Phil}_0 || \text{Stick}_4$$
Deadlock (Con.)

- This design leads to a deadlock situation if all philosophers pick up their left stick at the same time.
  - The corresponding execution leads from the initial state:
    
    \(<\text{think}_4, \text{avail}_3, \text{think}_3, \text{avail}_2, \text{think}_2, \text{avail}_1, \text{think}_1, \text{avail}_0, \text{think}_0, \text{avail}_4>\)

  by action sequence request\(_4\), request\(_3\), request\(_2\), request\(_1\), request\(_0\) to the terminal state:

  \(<\text{wait}_4, \text{occ}_4, \text{wait}_3, \text{occ}_3, \text{wait}_2, \text{occ}_2, \text{wait}_1, \text{occ}_1, \text{wait}_0, \text{occ}_0>\)
Deadlock(Con.)

- A possible solution to this problem is to make sticks available for only one philosopher at a time by changing stick model as follows:

- Why?
To analyze a computer system represented by a transition system two approaches can be followed:

- **State-based** approach: abstracts from actions and only state labels in sequences are considered.
- **Action-based** approach: abstracts from states and refers only to the action labels of the transitions.
Linear-Time Behavior (Con.)

- In this chapter, we mainly focus on the state-based approach.
  - Action labels of transitions are only necessary for modeling communication.
- Therefore, the verification algorithms operate on the state graphs of a LTS:
  - The digraph originating from a LTS by abstracting from action labels.
Linear-Time Behavior: Paths and State Graph

- Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a TS.
- **Definition:** The state graph of $TS$, notation $G(TS)$, is the digraph $(V,E)$ with vertices $V = S$ and edges $E = \{(s,s') \in S \times S \mid s' \in \text{Post}(s)\}$
- It should be noted that multiple transitions (with different labels) are represented by a single edge.
Let $\text{Post}^*(s)$ denote the states that are reachable in state graph $G(ST)$ from $s$.

This notation is generalized for sets of states in usual way: for $C \subseteq S$ let:

$$\text{Post}^*(C) = \bigcup_{s \in C} \text{Post}^*(s)$$

The notations $\text{Pre}^*(s)$ and $\text{Pre}^*(C)$ have analogous meaning.
The set of states that are reachable from some initial state, notation Reach(TS), equals Post*(I).

Since actions are abstracted, the resulting “runs” of a transition system are called paths.
**Definition:** A finite path fragment $\hat{\pi}$ of TS is a finite state sequence $s_0 s_1 \ldots s_n$ such that $s_i \in \text{Post}(s_{i-1})$ for all $0 < i \leq n$, where $n \geq 0$. An infinite path fragment $\pi$ is an infinite state sequence $s_0 s_1 s_2 \ldots$ such that $s_i \in \text{Post}(s_{i-1})$ for all $i > 0$. 
Linear-Time Behavior: Paths and State Graph (Con.)

- Some notations:
  - $\pi = s_0 \; s_1 \; \ldots$ : an infinite path fragment;
  - $\text{first}(\pi)$: the initial state of $\pi$;
  - $\pi[j] = s_j$, $j \geq 0$ : $j$th state of $\pi$;
  - $\pi[..j]$: the $j$th prefix of $\pi$, i.e., $\pi[..j] = s_0 \; s_1 \; \ldots \; s_j$;
  - $\pi[j..]$: the $j$th suffix of $\pi$, i.e., $\pi[j..] = s_j \; s_{j+1} \; \ldots$;
  - These notation are defined for finite paths:
  - $\hat{\pi} = s_0 \; s_1 \; \ldots \; s_n$ : an finite path fragment.
Linear-Time Behavior: Paths and State Graph (Con.)

- \( \text{last}(\hat{\pi}) = s_n \): the last state of \( \hat{\pi} \);
- \( \text{len}(\hat{\pi}) = n \): the length of \( \hat{\pi} \);
- For infinite path \( \pi \) we have \( \text{len}(\pi) = \infty \) and \( \text{last}(\pi) = \bot \), where \( \bot \) denotes “undefined”.
**Definition:** A maximal path fragment is either a finite path fragment that ends in a terminal state or an infinite path fragment. A path fragment is called initial if it starts in an initial state.

A maximal path fragment cannot be prolong: either it is infinite or finite but ends in a terminal state.
Linear-Time Behavior: Paths and State Graph (Con.)

- **Definition**: A path of transition system TS is an initial, maximal path fragment.
Linear-Time Behavior: Paths and State Graph (Con.)

- Some notations:
  - Paths(s): the set of maximal path fragments \( \pi \) with first(\( \pi \))=s;
  - Paths\(_{\text{fin}}\)(s): the set of all finite path fragments \( \hat{\pi} \) with first(\( \hat{\pi} \))=s;
  - Paths(TS): the set of all paths in TS;
  - Paths\(_{\text{fin}}\)(TS): the set of all initial, finite path fragments of TS.
Example 3.7: consider the beverage vending machine of Example 2.2.

Recall $L(s) = \{s\}$. Example path fragments of this TS are:

- $\pi_1 = \text{pay select tea pay select tea...}$
- $\pi_2 = \text{select tea pay select coffee...}$
- $\hat{\pi} = \text{pay select tea pay select tea}$. 
Linear-Time Behavior: Paths and State Graph (Con.)

- Only $\pi_1$ is a path. The infinite path fragment $\pi_2$ is maximal but not initial. $\hat{\pi}$ is initial but not maximal.

- We have last($\hat{\pi}$) = tea, first($\pi_2$) = select, $\pi_1[0]$ = pay, $\pi_1[3]$ = pay, $\pi_1[..5]$ = $\hat{\pi}$, $\hat{\pi}[..2]$ = $\hat{\pi}[3..]$, len($\hat{\pi}$) = 5, and len($\pi_1$) = $\infty$. 
Linear-Time Behavior: Traces

- We are going to focus on states visited during executions (i.e. paths) to analyze the behavior of a computer system.

- In fact states themselves are not observable, but just their atomic propositions.
Linear-Time Behavior : Traces (Con.)

- Thus instead of $s_0 \ s_1 \ s_2 \ ...$ we consider sequences of the form $L(s_0) \ L(s_1) \ L(s_2) \ ...$ that registers the atomic propositions that are valid along the execution.
- Such sequences are called traces.
- The traces of a transition system are thus words over the alphabet $2^{AP}$. 
In the following it is assumed that a transition system has no terminal state. In this case, all traces are infinite words. This assumption is made for simplicity and does not impose restriction!
Linear-Time Behavior : Traces (Con.)

- Thus prior to checking any property, a reachability analysis is carried out to determine the set of terminal states:
  - A terminal state may be a deadlock and has to be repaired before any further analysis;
Or it is a valid terminal state, and the TS can be extended such that:

- a new state $s_{\text{stop}}$ and transition $s \rightarrow s_{\text{stop}}$ and $s_{\text{stop}}$ is equipped with a self loop, i.e. $s_{\text{stop}} \rightarrow s_{\text{stop}}$

- The resulting equivalent transition system obviously has no terminal state.
Linear-Time Behavior: Traces (Con.)

**Definition:** Let $TS=(S,\text{Act},\rightarrow,I,\text{AP},L)$ be a TS without terminal states. The trace of the infinite path fragment $\pi = s_0 s_1 ...$ is defined as $\text{trace}(\hat{\pi}) = L(s_0) L(s_1) ...$. The trace of the finite path fragment $\hat{\pi} = s_0 s_1 ... s_n$ is defined as $\text{trace}(\hat{\pi}) = L(s_0) L(s_1) ... L(s_n)$. 


The trace of a path fragment is the sequence of sets of atomic propositions that are valid in the states of the path. The set of traces of a set $\Pi$ of paths is defined by $\text{trace}(\Pi) = \{\text{trace}(\pi) | \pi \in \Pi\}$. 

Read Example 3.9.
Let $\text{Traces}(s)$ denote the set of traces of $s$ and $\text{Traces}(\text{TS})$ the set of traces of transition system $\text{TS}$:

$$\text{Traces}(s) = \text{trace}(\text{Paths}(s)), \text{Traces}(\text{TS}) = \bigcup_{s \in I} \text{Traces}(s)$$

Similarly, the finite traces of a state $s$ and a transition system are defined:

$$\text{Traces}_{\text{fin}}(s) = \text{trace}(\text{Paths}_{\text{fin}}(s)),$$

$$\text{Traces}_{\text{fin}}(\text{TS}) = \bigcup_{s \in I} \text{Traces}_{\text{fin}}(s)$$
Linear-Time Behavior: Linear-Time Properties

- Linear-time properties specify the traces that a transition system should exhibit.
- Informally a linear-time property specifies the admissible behavior of the system under consideration.
Linear-Time Behavior: Linear-Time Properties (Con.)

- A LT property is a requirement on the traces of a TS (all words over AP), and is defined as the set of words that are admissible:

- **Definition**: A linear-time property (LT property) over the set of atomic proposition AP is a subset of $(2^{AP})^\omega$. 
Linear-Time Behavior: Linear-Time Properties (Con.)

- **Definition:** Let $P$ be an LT property over $AP$ and $TS=(S,Act,\rightarrow,I,AP,L)$ a transition system without terminal state. Then $TS=(S,Act,\rightarrow,I,AP,L)$ satisfies $P$, denoted $TS \models P$, iff $\text{Traces}(TS) \subseteq P$. States $s \in S$ satisfies $P$, notation $s \models P$, whenever $\text{Traces}(s) \subseteq P$.  


Linear-Time Behavior: Linear-Time Properties (Con.)

- Thus a transition system satisfies the LT property $P$ if all its traces respect $P$, i.e., if all its behavior are admissible.
- A state satisfies $P$ whenever all traces starting in this state fulfill $P$. 
Example 3.12: Consider two simplified traffic lights that only have two possible settings: red and green. Let the proposition of interest be

\[ AP=\{\text{red}_1,\text{green}_1,\text{red}_2,\text{green}_2\} \]
Linear-Time Behavior: Linear-Time Properties (Con.)

- Consider property $P$ that states “The first traffic light is infinitely often green”

- $P$ corresponds to the set of infinite words of the form $A_0 A_1 A_2 \ldots$ over $2^{AP}$, such that $\text{green}_1 \in A_i$ holds for infinitely many $i$.

- For example $P$ contains the infinite words
  
  - $\{\text{red}_1, \text{green}_2\} \{\text{green}_1, \text{red}_2\} \{\text{red}_1, \text{green}_2\} \{\text{green}_1, \text{red}_2\} \ldots$
  - $\emptyset \{\text{green}_1\} \emptyset \{\text{green}_1\} \emptyset \{\text{green}_1\} \emptyset \{\text{green}_1\} \emptyset \ldots$
  - $\{\text{red}_1, \text{green}_1\} \{\text{red}_1, \text{green}_1\} \{\text{red}_1, \text{green}_1\} \{\text{red}_1, \text{green}_1\} \ldots$

  - **But** $\{\text{red}_1, \text{green}_1\} \{\text{red}_1, \text{green}_1\} \emptyset \emptyset \emptyset \emptyset \ldots$ is not in $P$. 
Consider property $P'$ that states “The traffic lights are never both green simultaneously”.

$P'$ is formalized by the set of infinite words of the form $A_0 A_1 A_2 ...$ such that either $\text{green}_1 \notin A_i$ or $\text{green}_2 \notin A_i$, for all $i \geq 0$.

For example $P'$ contains:

- $\{\text{red}_1, \text{green}_2\}\{\text{green}_1, \text{red}_2\}\{\text{red}_1, \text{green}_2\}\{\text{green}_1, \text{red}_2\}...$
- $\emptyset\{\text{green}_1\}\emptyset\{\text{green}_1\}\emptyset\{\text{green}_1\}\emptyset\{\text{green}_1\}\emptyset...$
- $\{\text{red}_1, \text{green}_1\}\{\text{red}_1, \text{green}_1\}\{\text{red}_1, \text{green}_1\}\{\text{red}_1, \text{green}_1\}...$

- **But** $\{\text{red}_1, \text{green}_2\}\{\text{green}_1, \text{green}_2\}...$ is not in $P'$. 
Linear-Time Behavior: Linear-Time Properties (Con.)

- The traffic lights depicted below satisfies both P and P', since their switching is synchronized:

  ![Traffic light diagram]

- Traffic lights that switch completely autonomously with neither satisfy P nor P'.
Linear-Time Behavior: Linear-Time Properties (Con.)

- Often, an LT property does not refer to all atomic propositions occurring in a TS, but just to a small subset.

- Let $P$ be a property over $AP' \subseteq AP$ such that only the labels in $AP'$ are relevant:
  - $\text{trace}_{AP'}(\hat{\pi})$ denotes the finite trace of $\hat{\pi}$ where only propositions in $AP'$ are considered.
Linear-Time Behavior: Linear-Time Properties (Con.)

- $\text{trace}_{AP'}(\pi)$ denotes the trace of an infinite path fragment $\pi$ by focusing on propositions in $AP'$, defined as:
  \[
  \text{trace}_{AP'}(\pi) = \text{L}'(s_0) \text{L}'(s_1) \ldots \\
  = (L(s_0) \cap AP') (L(s_1) \cap AP') \ldots
  \]

- $\text{Traces}_{AP'}(TS)$ denote the set of traces $\text{trace}_{AP'}(\text{Paths}(TS))$.

- Whenever the set $AP'$ is clear from the context, the subscript $AP'$ is omitted.
Example 3.13: For specifying the mutual exclusion, it suffices to only consider the atomic proposition $\text{crit}_1$ and $\text{crit}_2$.

Thus the formalization of mutual exclusion property is:

$$P_{\text{mutex}} = \text{set of infinite words } A_0 A_1 A_2 \ldots \text{ with } \{\text{crit}_1, \text{crit}_2\} \not\subseteq A_i \text{ for all } i \geq 0.$$
Linear-Time Behavior: Linear-Time Properties (Con.)

- For example $P_{\text{mutex}}$ contains:
  
  \[
  \{\text{crit}_1\}\{\text{crit}_2\}\{\text{crit}_1\}\{\text{crit}_2\}\{\text{crit}_1\}\{\text{crit}_2\}... \\
  \{\text{crit}_1\}\{\text{crit}_1\}\{\text{crit}_1\}\{\text{crit}_1\}\{\text{crit}_1\}\{\text{crit}_1\}... \\
  \varnothing\varnothing\varnothing\varnothing\varnothing... \\
  \]

- But $\{\text{crit}_1\}\varnothing\{\text{crit}_1,\text{crit}_2\}...$ is not in $P_{\text{mutex}}$.

- The transition system $TS_{\text{Arb}}$ = $(TS_1 \parallel TS_2) \parallel TS_{\text{Arb}}$ in Example 2.28 fulfills the mutex property: $TS_{\text{Arb}} \models P_{\text{mutex}}$.

- Read Example 3.14 for starvation freedom!
Linear-Time Behavior: Trace Equivalence and Linear-Time Properties

- LT properties specify the (infinite) traces that a TS should exhibit.
- If TS and TS’ have the same traces, one would expect that they satisfy the same LT properties:
  - If TS |= P and Traces(TS)=Traces(TS’), then TS’ |= P, the traces of TS’ are also contained in P.
Linear-Time Behavior: Trace Equivalence and Linear-Time Properties (Con.)

- Whenever $TS \not\equiv P$, then there is a trace in $\text{Traces}(TS)$ that is prohibited by $P$.
- If $\text{Traces}(TS) = \text{Traces}(TS')$, also $TS'$ exhibits this prohibited trace and thus $TS' \not\equiv P$.
- In this Section we define the relationship between trace equivalence, trace inclusion and satisfaction of LT properties.
Linear-Time Behavior: Trace Equivalence and Linear-Time Properties (Con.)

- **Trace inclusion** between transition systems TS and TS’ requires that:
  - all traces exhibited by TS can also be exhibited by TS’: $\text{Traces(TS)} \subseteq \text{Traces(TS')}$.  
  
  TS’ may exhibit more traces, i.e., may have some behavior that TS does not have.
In stepwise design, where designs are successively refined, trace inclusion is viewed as implementation relation:

- Traces(TS) ⊆ Traces(TS’), means TS “is a correct implementation of” TS’.
- For example, TS’ may be a design where parallel composition is modeled by interleaving and TS its realization where interleaving is resolved by some scheduling.
Theorem 3.15 (trace inclusion and LT properties): Let TS and TS’ be transition systems without terminal states and with the same set of AP. Then the following statements are equivalent:

- $\text{Traces}(TS) \subseteq \text{Traces}(TS')$.
- For any LT property P: $TS' \models P$ implies $TS \models P$. 
This Theorem plays a decisive role for design by means of successive refinement:

- If TS’ is the transition system representing a preliminary design and TS its refinement, it can immediately be concluded that any LT property that holds in TS’ also holds for TS.
- Read Example 3.16.
**Definition**: Transition systems $TS$ and $TS'$ are **trace-equivalent** with respect to the set of propositions $AP$ if $\text{Traces}_{AP}(TS)=\text{Traces}_{AP}(TS')$.

**Corollary**: Let $TS$ and $TS'$ be transition systems without terminal state and with the same set of propositions. Then $\text{Traces}(TS)=\text{Traces}(TS') \iff TS$ and $TS'$ satisfy the same LT properties.
Thus there does not exist an LT property that can distinguish between trace-equivalent TSs.

So to establish TS and TS’ are not trace-equivalent, it suffices to find one LT property that holds for one but not for the other.
Example 3.19: Consider the two transition systems below:

Let \( \text{AP} = \{\text{pay}, \text{soda}, \text{beer}\} \). Thus they are trace-equivalent.
Safety Properties and Invariants

- **Safety** properties are often characterized as “nothing bad should happen”:
  - Mutual exclusion property;
  - Deadlock freedom.
The above safety properties are of a particular kind: they are **invariants**:

- Invariants are LT properties that are given by a condition \( \Phi \) for the states and requires that \( \Phi \) holds for all reachable states.
Safety Properties and Invariants (Con.)

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

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

  $\Phi$ is called an invariant condition (or state condition) of $P_{\text{inv}}$. 
Safety Properties and Invariants: Invariants (Con.)

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

- The condition $\Phi$ has to be fulfilled by all initial states and satisfaction of $\Phi$ is invariant under all transitions:
  - If $s \models \Phi$ and $s$ $\xrightarrow{\alpha} s'$, then $s' \models \Phi$. 
Safety Properties and Invariants: Invariants (Con.)

- The mutual property can be described by an invariant using the propositional logic formula $\Phi = \neg \text{crit}_1 \lor \neg \text{crit}_2$.

- For deadlock freedom of the dining philosophers, the invariant ensures that at least one of the philosophers is not waiting to pick up the chopstick:

  $\Phi = \neg \text{wait}_0 \lor \neg \text{wait}_1 \lor \neg \text{wait}_2 \lor \neg \text{wait}_3 \lor \neg \text{wait}_4$. 
Safety Properties and Invariants: Invariants (Con.)

- Checking an invariant for the propositional formula $\Phi$ in a TS amounts to checking the validity of $\Phi$ in every reachable state:
  - A slight modification of standard graph traversal algorithms like depth-first search (DFS) or breath-first search (BFS), provided that the TS is finite.
Safety Properties and Invariants: Invariants (Con.)

- The algorithm below checks the invariant condition $\Phi$ by means of a forward DFS in the state graph $G(TS)$:

```plaintext
set of state $R := \emptyset$;
stack of state $U := \varepsilon$;
bool $b :=$ true;
for all $s \in I$ do
  if $s \notin R$ then
    visit($s$)
  fi
od
return $b$

procedure visit (state $s$)
  push($s$, $U$);
  $R := R \cup \{ s \}$;
  repeat
    $s' :=$ top($U$);
    if $Post(s') \subseteq R$ then
      pop($U$);
      $b := b \land (s' \models \Phi)$;
    else
      let $s'' \in Post(s') \setminus R$
      push($s''$, $U$);
      $R := R \cup \{ s'' \}$;
    fi
  until ($U = \varepsilon$)
endproc
```
Forward search means that we start from the initial state and investigates all states that are reachable from them.

R stores all visited states, so when the algorithm terminates R=Reach(TS).

U is a stack that organized all states that still have to be visited, provided that they are not yet in R.
This algorithm could be improved by
- aborting the computation once a state $s$ is encountered that does not fulfill $\Phi$.
- While a counterexample is returned on encountering a state that violates $\Phi$. 
Safety Properties and Invariants: Invariants (Con.)

- The resulting algorithm is shown:

```plaintext
set of states $R := \emptyset$;
stack of states $U := \varepsilon$;
bool $b := true$;
while ($I \setminus R \neq \emptyset \land b$) do
  let $s \in I \setminus R$;
  visit($s$);
  od
if $b$ then
  return(”yes”)
else
  return(”no”, reverse($U$))
fi
```

```plaintext
procedure visit (state $s$)
  push($s$, $U$);
  $R := R \cup \{s\}$;
  repeat
    $s' := \text{top}(U)$;
    if $\text{Post}(s') \subseteq R$ then
      pop($U$);
      $b := b \land (s' \models \Phi)$;
    else
      let $s'' \in \text{Post}(s') \setminus R$
      push($s''$, $U$);
      $R := R \cup \{s''\}$;
    fi
  until ($U = \varepsilon$) \lor \neg b
endproc
```

- Examine the time complexity of algorithm!
Time Complexity of Invariant Checking

**Theorem:** The time complexity of Algorithm 4 is $O(N^*(1+|Φ|) + M)$ where $N$ denotes the number of reachable states, and $M$ the number of transitions in the reachable fragment of TS.
Alternate to a forward search, a **backward search** could have been applied that starts with all states where $\Phi$ does not hold and calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \notin \Phi} \text{Pre}^*(s)$.
Safety Properties and Invariants: Safety Properties

- Invariants can be viewed as state properties and can be checked by considering the reachable states.

- Some safety properties may impose requirement on finite path fragment and cannot be verified by considering the reachable states only.
A requirement for an automated teller machine (ATM) is that money can be withdrawn from the dispenser once a correct PIN has been provided.

- This property is not an invariant, since it is not a state property.
- It is a safety property as any infinite run violating the requirement has a finite prefix that is bad – money is withdrawn without PIN.
Safety Properties and Invariants: Safety Properties (Con.)

- Formally, a safety property $P$ is defined as an LT property over $AP$ such that any infinite word $\sigma$ where $P$ does not hold contains a bad prefix.
Safety Properties and Invariants: Safety Properties (Con.)

- **Definition**: An LT property $P_{\text{safe}}$ over $\mathcal{AP}$ is called a safety property if for all words $\sigma \in (2^{\mathcal{AP}})^{\omega} \setminus P_{\text{safe}}$ there exists a finite path $\hat{\sigma}$ of $\sigma$ such that

$$P_{\text{safe}} \cap \{ \sigma' \in (2^{\mathcal{AP}})^{\omega} | \hat{\sigma} \text{ is a finite prefix of } \sigma' \} = \emptyset$$
Any such finite word $\hat{\sigma}$ is called a bad prefix for $P_{\text{safe}}$. A minimal bad prefix $P_{\text{safe}}$ denotes a bad prefix $\hat{\sigma}$ for $P_{\text{safe}}$ for which no proper prefix of $\hat{\sigma}$ is a bad prefix for $P_{\text{safe}}$.

- In other words, minimal bad prefixes are bad prefixes of minimal length.

- $\text{BadPref}(P_{\text{safe}})$ denotes the set of all bad prefixes for $P_{\text{safe}}$ and $\text{MinBadPref}(P_{\text{safe}})$ the set of all minimal bad prefixes.
Safety Properties and Invariants: Safety Properties (Con.)

- Any invariant is a safety property.
  - For propositional formula $\Phi$ over $\text{AP}$ and its invariant $P_{inv}$, all finite words of the form $A_0 \ A_1 \ldots \ A_n \in (2^{\text{AP}})^+$ with $A_0 \models \Phi$, ..., $A_{n-1} \models \Phi$ and $A_n \not\models \Phi$ constitute the minimal bad prefixes for $P_{inv}$. 
Example 3.23: We consider a specification of a traffic light with the usual three phases red, green, and yellow.

The requirement that each red phase should be immediately preceded by a yellow phase is a safety property but not an invariant.
Safety Properties and Invariants: Safety Properties (Con.)

- Let \( \text{AP} = \{\text{red, green, yellow}\} \).
- This property is specified by the set of infinite words \( \sigma = A_0 A_1 \ldots \) with \( A_i \subseteq \{\text{red, yellow}\} \) such that for all \( i \geq 0 \) we have that
  - \( \text{red} \in A_i \) implies \( i > 0 \) and \( \text{yellow} \in A_{i-1} \).
  - The bad prefixes are finite words that violates this condition.
Safety Properties and Invariants: Safety Properties (Con.)

- The minimal bad prefixes of this property constitute a regular language accepted by the below automaton:

See also Example 3.24.
Safety Properties and Invariants: Safety Properties (Con.)

- **Lemma (Satisfaction Relation for Safety Properties):** For transition system TS without terminal state and safety property $P_{safe}$:
  - $TS \models P_{safe}$ if and only if $\text{Traces}_{fin}(TS) \cap \text{BadPref}(P_{safe}) = \emptyset$. 
Safety Properties and Invariants: Safety Properties (Con.)

**Definition:** For trace $\sigma \in (2^{AP})^\omega$, let $\text{pref}(\sigma)$ denote the set of finite prefixes of $\sigma$:

- $\text{pref}(\sigma) = \{ \hat{\sigma} \in (2^{AP})^* \mid \hat{\sigma} \text{ is a finite prefix of } \sigma \}$.
- If $\sigma = A_0A_1...$ then $\text{pref}(\sigma) = \{ \varepsilon, A_0, A_0A_1, A_0A_1A_2, ... \}$ is an infinite set of finite words.
- This notion is lifted to sets of traces in usual way. For property $P$: $\text{pref}(P) = \bigcup_{\sigma \in P} \text{pref}(\sigma)$
The closure of LT property $P$ is the set of infinite traces whose finite prefix are also prefixes of $P$.

Stated differently, infinite traces in the closure of $P$ do not have a prefix that is not a prefix of $P$ itself.
Safety Properties and Invariants: Safety Properties (Con.)

- **Lemma (Alternative Characterization of Safety Properties):** Let $P$ be an LT property over $AP$. Then, $P$ is a safety property iff $\text{closure}(P) = P$. 
Theorem (Finite Trace Inclusion and Safety Properties): Let TS and TS’ be transition systems without terminal states and with a same AP. Then the following statements are equivalent:

- \( \text{Traces}_{\text{fin}}(TS) \subseteq \text{Traces}_{\text{fin}}(TS') \);
- For any safety property \( P_{\text{safe}} : TS' |= P_{\text{safe}} \) implies \( TS |= P_{\text{safe}} \).
Safety Properties and Invariants: Trace Equivalence and Safety Properties (Con.)

- Thus if a preliminary design $TS'$ is refined to a design $TS$ such that
  - $\text{Traces}(TS) \not\subseteq \text{Traces}(TS')$ then the LT properties of $TS'$ do not hold for $TS$.
  - $\text{Traces}_{\text{fin}}(TS) \subseteq \text{Traces}_{\text{fin}}(TS')$ (which is a weaker requirement than full trace inclusion of $TS$ and $TS'$) then all safety properties of $TS'$ also hold for $TS$. 


Safety Properties and Invariants: Trace Equivalence and Safety Properties (Con.)

**Corollary:** Let TS and TS’ be transition systems without terminal states and with a same AP. Then the following statements are equivalent:

- \( \text{Traces}_{\text{fin}}(TS) = \text{Traces}_{\text{fin}}(TS') \);
- For any safety property \( P_{\text{safe}} \) over AP: \( TS' |= P_{\text{safe}} \iff TS |= P_{\text{safe}} \).
Safety Properties and Invariants: Trace Equivalence and Safety Properties (Con.)

- Since we assume transition systems without terminal states, there is only a slight difference between trace inclusion and finite state inclusion.
  - For finite transition systems TS and TS’ without terminal states, trace inclusion and finite trace inclusion coincide.
Theorem (Relating Finite Trace and Trace Inclusion): Let TS and TS’ be transition systems with the same set AP of atomic propositions such that TS has no terminal states and TS’ is finite. Then

\[ \text{Traces}(TS) \subseteq \text{Traces}(TS’) \iff \text{Traces}_{\text{fin}}(TS) \subseteq \text{Traces}_{\text{fin}}(TS’); \]
Remark: The result of previous Theorem also holds under weaker conditions: TS has no terminal states and TS’ is AP image-finite.

TS’ is called AP image-finite if

- For all $A \subseteq \text{AP}$, the set $\{s_0 \in I \mid L(s_0) = A\}$ is finite.
- For all state $s$ in TS’ and $A \subseteq \text{AP}$, the set of successors $\{s' \in \text{Post}(s) \mid L(s') = A\}$ is finite.
Liveness Properties

- Liveness properties state that “something good” will happen in the future.

- Whereas safety properties are violated in finite time, i.e., by a finite system run, liveness properties are violated in infinite time, i.e., by infinite system runs.
Definition: LT property $P_{\text{live}}$ over AP is a liveness property whenever $\text{pref}(P_{\text{live}}) = (2^{\text{AP}})^*$. Thus a liveness property is an LT property such that each finite word can be extended to an infinite word that satisfies $P$. 
Liveness Properties (Con.)

- Example 3.34: In the context of mutual exclusion, typical liveness properties are
  - (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.

- Formalize above properties!
Liveness Properties: Safety vs. Liveness Properties

- We examine following questions:
  - Are safety and liveness properties disjoint?
  - Is any linear-time property a safety or liveness property?
Liveness Properties: Safety vs. Liveness Properties (Con.)

- Safety and liveness properties are indeed disjoint:
- **Lemma 3.35 (Intersection of Safety and Liveness Properties):** The only LT property over AP that is both a safety and a liveness property is \((2^{AP})^\omega\).
Liveness Properties: Safety vs. Liveness Properties (Con.)

- For any LT property $P$ an equivalent LT property $P'$ does exist which is a combination (i.e., intersection) of a safety and a liveness property.

- Lemma 3.36 (Distributivity of Union over Closure): for any LT properties $P$ and $P'$:
  
  $$\text{closure}(P) \cup \text{closure}(P') = \text{closure}(P \cup P')$$
Theorem 3.37 (Decomposition Theorem): For any LT property $P$ over $AP$ there exists a safety property $P_{safe}$ and a liveness property $P_{live}$ (both over $AP$) such that $P = P_{safe} \cap P_{live}$. 

- The proof of this theorem shows that $P_{safe} = \text{closure}(P)$ and $P_{live} = P \cup ((2_{AP})^\omega \setminus \text{closure}(P))$. 

Liveness Properties: Safety vs. Liveness Properties (Con.)

- This decomposition is the sharpest since $P_{\text{safe}}$ is the strongest safety property and $P_{\text{live}}$ the weakest liveness property.

**Lemma 3.38 (Sharpest Decomp.):**

Let $P$ be a LT property and $P = P_{\text{safe}} \cap P_{\text{live}}$ where $P_{\text{safe}}$ is a safety property and $P_{\text{live}}$ a liveness property. We have

- $\text{closure}(P) \subseteq P_{\text{safe}}$;
- $P_{\text{live}} \subseteq P \cup (\mathcal{F}(\mathcal{A})^\omega \setminus \text{closure}(P))$. 

98
Liveness Properties: Safety vs. Liveness Properties (Con.)

- A summary of the classification of LT properties is depicted as a Venn diagram in below:

![Venn Diagram](image-url)
Fairness

- Fairness assumptions rule out infinite behaviors that are considered unrealistic, and are often necessary to establish liveness property.

- When verifying concurrent systems we are interested in paths in which enabled transitions are executed in some fair manner.
In order to prove starvation freedom, we want to exclude those paths in which the competitor process is always being selected for execution.

This type of fairness is also known as process fairness, since it concerns the fair scheduling of the execution of processes.
In general, fairness assumptions are needed to prove liveness or other properties stating that the system makes some progress.

Besides it is important if the TS contains nondeterminism.

Fairness is used to resolve nondeterminism in modeling concurrent processes by means of interleaving.
Example 3.42: Consider the transition system $TS = \text{TrLight}_1 \ || \ \text{TrLight}_2$ for the two independent traffic lights.

The liveness property “both traffic lights are infinitely often green” is not satisfied, since \{red1,red2\} \{green1,red2\} \{red1,red2\} \{green1,red2\} \ldots is a trace of $TS$. 
Fairness (Con.)

- The problem is that the information that each light switches color infinitely often is lost by means of interleaving.
- Thus the trace in which only the first light is acting while the second seems to be stopped is formally a trace of TS.
- However, it does not represent a realistic behavior as in practice no light is infinitely faster than another.
To obtain a realistic picture of the behavior of a parallel system modeled by a TS, we need to resolve nondeterministic decisions in a TS.

In order to rule out the unrealistic computations, fairness constraints are imposed.
A fair execution is characterized by the fact that certain fairness constraints are fulfilled.

Fairness constraints are used to rule out computations that are considered to be unreasonable.
Fairness: Fairness Constraints (Con.)

- Let “is enabled” mean TS “is ready to execute (a transition)” and “gets its turn” stand for the execution of an arbitrary transition.
Fairness: Fairness Constraints (Con.)

- Fairness constraints are of three types:
  - **Unconditional** fairness: e.g., “every process gets its turn infinitely often.”
  - An execution fragment is unconditionally fair with respect to e.g. “a process enters its critical section” or “a process gets its turn”, if these properties hold infinitely often.
Fairness: Fairness Constraints (Con.)

- **Strong** fairness: e.g., “every process that is enabled infinitely often gets its turn infinitely often”.
  - An execution fragment is strongly fair with respect to activity $\alpha$ if it is not the case that $\alpha$ is infinitely enabled without being taken beyond a certain point.
Fairness: Fairness Constraints (Con.)

- **Weak fairness**: e.g., “every process that is continuously enabled from a certain time instant on gets its turn infinitely often”.
  
  - An execution fragment is weakly fair with respect to some activity, say $\alpha$, if it is not the case that $\alpha$ is always enabled beyond some point without being taken beyond this point.
There are different ways to formulate fairness requirements.

In following, we adopt the action-based view and define fairness for sets of actions.

In order to formulate fairness notions formally, the following auxiliary notion is used:

Act(s) denotes the set of actions that are executable in state s, that is,

\[ \{a \in \text{Act} | \exists s' \in S, \ s \xrightarrow{\alpha} s' \} \]
Fairness: Fairness Constraints (Con.)

**Definition:** For $TS= (S, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots$ of $TS$:

1. $\rho$ is **unconditionally** $A$-fair whenever $\exists^\infty j. \alpha_j \in A$.
2. $\rho$ is **strongly** $A$-fair whenever $(\exists^\infty j. Act(s_j) \cap A \neq \emptyset) \Rightarrow (\exists^\infty j. \alpha_j \in A)$.
3. $\rho$ is **weakly** $A$-fair whenever $(\forall^\infty j. Act(s_j) \cap A \neq \emptyset) \Rightarrow (\exists^\infty j. \alpha_j \in A)$. 
Fairness: Fairness Constraints (Con.)

Here $\exists^\infty j$ stands for “there are infinitely many $j$” and $\forall^\infty j$ for “for nearly all $j$” in the sense of “for all, except for finitely many $j$”. The variable $j$ ranges over the natural numbers.
To check whether a run is unconditionally A-fair it suffices to consider the actions that occur along the execution.

However to check whether a given execution is strongly or weakly A-fair, we should also consider the enabled actions in all visited states.
Example 3.44. Consider the following two processes that run in parallel and share variable x:

- Proc Inc=while \( x \geq 0 \) do \( x := x + 1 \) od where the pair of \( \langle \ldots \rangle \) embraces an atomic action.
- Proc Reset= \( x := -1 \)

The termination is not guaranteed for this parallel program. If, however, we require unconditional process fairness, termination is guaranteed.
The relationship between different fairness notions:

- Each unconditionally A-fair execution fragment is strongly A-fair;
- Each strongly A-fair execution fragment is weakly A-fair.
- Read Example 3.45.
A fairness constraint imposes a requirement on all actions in a set $A$. We can use fairness assumptions to have fairness constraints on different, possibly nondisjoint, sets of actions.
Definition: A fairness assumption for Act is a triple $F= (F_{\text{uncond}}, F_{\text{strong}}, F_{\text{weak}})$ with $F_{\text{uncond}}, F_{\text{strong}}, F_{\text{weak}} \subseteq 2^{\text{Act}}$. Execution $\rho$ is F-fair if:

- It is unconditionally A-fair for all $A \in F_{\text{uncond}},$
- It is strongly A-fair for all $A \in F_{\text{strong}}$, and
- It is weakly A-fair for all $A \in F_{\text{weak}}$.

If the set F is clear from the context, we use the term fair instead of F-fair.
The notion of F-fairness as defined on execution fragments is lifted to traces and paths:

- An infinite trace $\sigma$ is F-fair if there is an F-fair execution $\rho$ with $\text{trace}(\rho)=\sigma$.
- F-fair path fragments and F-fair paths are defined analogously.
Let $\text{FairPaths}_F(s)$ denote the set of $F$-fair paths of $s$ and $\text{FairPaths}_F(TS)$ the set of $F$-fair paths that start in some initial state of $TS$.

Let $\text{FairTraces}_F(s)$ denote the set of $F$-fair traces of $s$ and $\text{FairTraces}_F(TS)$ the set of $F$-fair traces of the initial states of $TS$:

$\text{FairTraces}_F(s) = \text{trace}(\text{FairPaths}_F(s))$ and $\text{FairTraces}_F(TS) = \bigcup_{s \in I} \text{FairTraces}_F(s)$. 

120
Example 3.47: consider the following fairness requirement for two-process mutual exclusion algorithm: “process Pi acquires access to its critical section (CR) infinitely often” for any $i \in \{1,2\}$.

- Assume $P_i$ has three states: $n_i$ (noncritical), $w_i$ (waiting), and $c_i$ (critical).
- Actions $req_i$, $enter_i$, and $rel$ to model request to enter CR, entering CR and release of the CR.
The strong-fairness assumption \( \{\text{enter}_1, \text{enter}_2\} \) ensures that one of the actions \( \text{enter}_1 \) or \( \text{enter}_2 \) is executed infinitely often.

A behavior in which one of the processes gets access to the CR infinitely often while the other gets access only finitely many times is strongly fair with respect to this assumption!
This is, however, not intended.
The strong-fairness assumption \{\text{enter1}, \text{enter2}\} indeed realizes the earlier.
Fairness: Fairness Constraints (Con.)

- It should be noted that fairness assumptions can be verifiable properties whenever all infinite execution fragments are fair.

- But in many cases it is necessary to assume the validity of the fairness to verify liveness properties.

- Read Examples 3.49 and 3.50.
Definition (Fair Satisfaction Relation for LT properties): Let $P$ be an LT property over $AP$ and $F$ a fairness assumption over $Act$. $TS=(S,Act,\rightarrow,I,AP,L)$ fairly satisfies $P$, notation $TS \models_F P$, if and only if $\text{FairTraces}_F(TS) \subseteq P$. 
In order to rule out "unrealistic" computations, fairness assumptions are imposed on the traces of TS and it is checked whether \( TS \models_F P \) as opposed to checking \( TS \models P \).

What fairness assumption is suitable for synchronizing concurrent systems?
To have fair communication mechanism between various processes, by rule of thumb:

- **Strong fairness** is needed to obtain an adequate resolution of contention.
- **Weak fairness** suffices for sets of actions that represent the concurrent execution of independent actions (i.e. interleaving).
- Read Example 3.51.
Fairness: Fairness and Safety

- While fairness assumptions may be necessary to verify liveness properties, they are irrelevant for verifying safety properties.
- However, it is required fairness assumptions always be ensured by means of an scheduling strategy.
  - Such fairness assumptions are called realizable fairness assumptions.
Definition: Let TS be a transition system with the set of actions Act and F a fairness assumption for Act. F is called realizable for TS if for every reachable state s: \( \text{FairPaths}_F(s) \neq \emptyset \).

A fairness assumption is realizable in a TS whenever in any reachable state at least one fair execution is possible.
Theorem 3.55 (Realizable Fairness is Irrelevant for Safety Properties)

Let TS be a transition system with set of propositions AP, F a realizable fairness assumption for TS, and \( P_{\text{safe}} \) a safety property over AP. Then:

\[ TS \models P_{\text{safe}} \iff TS \models F P_{\text{safe}}. \]

Theorem does not hold if arbitrary fairness assumptions are permitted (Example 3.56).
Summary

- The set of reachable states of a TS can be determined by a search algorithm on the state graph of TS.

- A trace is a sequence of sets (!) of atomic propositions. The traces of a TS are obtained from projecting the paths to the sequence of state labels.

- A linear-time property is a set of infinite words over the alphabet $2^{AP}$. 
Summary

- Two TSs are **trace-equivalent** iff they satisfy the **same LT properties**.
- An **invariant** is an LT property that is **purely state-based** and requires a **propositional** logic formula $\Phi$ to hold for all reachable states. Invariants can be checked using a **DFS** where the DFS stack can be used to provide a counter-example in case an invariant is refuted.
Summary

- Safety properties are generalizations of invariants. They constrain the finite behaviors. The formal definition of safety properties can be provided by means of their bad prefixes in the sense that each trace that refutes a safety property has a finite prefix, the bad prefix, that causes this.
Two TSs exhibit the same finite traces iff they satisfy the same safety properties.

A liveness property is an LT property if it does not rule out any finite behavior. It constrains infinite behavior.

Any LT property is equivalent to an LT property that is a conjunction of a safety and aliveness property.
Fairness assumptions serve to rule out traces that are considered to be unrealistic. They consist of unconditional, strong, and weak fairness constraints, i.e., constraints on the actions that occur along infinite executions.
Summary

- Fairness assumptions are often necessary to establish liveness properties, but they are provided they are realizable irrelevant for safety properties.