Analysis of Reactive Systems

Amir Pnueli

Mondays, 5:00-6:50 PM

Copies of presentations and Lecture Notes will be available at
http://www.cs.nyu.edu/courses/spring06/G22.3033-05/index.htm

Textbooks:

Course Outline

The course will focus on formal verification of reactive systems. The course will be dedicated to methods for the verification of large systems. The main topics we will consider are:

- Systems and their Specification by LTL, CTL, and CTL*. model checking of finite-state systems using BDD techniques over the TLV tool.

- Deductive verification of infinite-state systems, using theorem provers such as PVS, and STeP.

- Abstraction methods for combining deductive principles with algorithmic methods.

Course grades will be determined based on assignments and a term project.
Classification of Programs

There are two classes of programs:

**Computational Programs:** Run in order to produce a final result on termination.

Can be modeled as a **black box.**

Specified in terms of **Input/Output** relations.

**Example:**

The program which computes

\[ y = 1 + 3 + \cdots + (2x - 1) \]

Can be specified by the requirement

\[ y = x^2. \]
Reactive Programs

Programs whose role is to maintain an ongoing interaction with their environments.

**Examples:** Air traffic control system, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor.

Can be viewed as a green cactus (?)

Such programs must be specified and verified in terms of their behaviors.
A Framework for Reactive Systems Verification

- A computational model providing an abstract syntactic base for all reactive systems. We use fair Discrete structures (FDS).

- A Specification Language for specifying systems and their properties. We use temporal logics: CTL*, CTL, and LTL.

- An Implementation Language for describing proposed implementations (both software and hardware). Use SPL, a simple programming language and the SMV input language for hardware systems description.

- Verification Techniques for validating that an implementation satisfies a specification. Practiced approaches:
  - A deductive methodology based on theorem-proving methods. Can accommodate infinite-state systems, but requires user interaction.
A fair discrete system (FDS) $D = \langle V, \Theta, \rho, J, C \rangle$ consists of:

- $V$ – A finite set of typed state variables. A $V$-state $s$ is an interpretation of $V$. $\Sigma_V$ – the set of all $V$-states.

- $\Theta$ – An initial condition. A satisfiable assertion that characterizes the initial states.

- $\rho$ – A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (current) and primed (next) versions of the state variables. For example, $x' = x + 1$ corresponds to the assignment $x := x + 1$.

- $J = \{J_1, \ldots, J_k\}$ A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many $J_i$-states for each $J_i$, $i = 1, \ldots, k$.

- $C = \{\langle p_1, q_1 \rangle, \ldots, \langle p_n, q_n \rangle\}$ A set of compassion (strong fairness) requirements. Infinitely many $p_i$-states imply infinitely many $q_i$-states.
A Simple Programming Language: SPL

A language allowing composition of parallel processes communicating by shared variables as well as message passing.

Example: Program ANY-Y

Consider the program

\[
x, y: \text{natural initially } x = y = 0
\]

\[
\begin{align*}
\ell_0 &: \text{ while } x = 0 \text{ do } \\
\ell_1 &: \quad y := y + 1 \\
\ell_2 &: \\
\end{align*}
\]

\[
\begin{align*}
m_0 &: \quad x := 1 \\
m_1 &:
\end{align*}
\]

\[
\begin{align*}
P_1 &- \quad - \\
P_2 &- \quad -
\end{align*}
\]
The Corresponding FDS

\[
\begin{align*}
V: & \quad \begin{pmatrix} x, y : & \text{natural} \\
\pi_1 : & \{\ell_0, \ell_1, \ell_2\} \\
\pi_2 : & \{m_0, m_1\}
\end{pmatrix} \\
\Theta : & \quad \pi_1 = \ell_0 \land \pi_2 = m_0 \land x = y = 0.
\end{align*}
\]

\[
\begin{align*}
\rho & : \quad \rho_I \lor \rho_{\ell_0} \lor \rho_{\ell_1} \lor \rho_{m_0}, \text{ with appropriate disjunct for each statement. For example, the disjuncts } \rho_I \text{ and } \rho_{\ell_0} \text{ are} \\
\rho_I : & \quad \pi'_1 = \pi_1 \land \pi'_2 = \pi_2 \land x' = x \land y' = y
\end{align*}
\]

\[
\begin{align*}
\rho_{\ell_0} : & \quad \pi_1 = \ell_0 \land \\
& \quad \left( x = 0 \land \pi'_1 = \ell_1 \lor x \neq 0 \land \pi'_1 = \ell_2 \right) \\
& \quad \land \pi'_2 = \pi_2 \land x' = x \land y' = y
\end{align*}
\]

\[
\begin{align*}
\mathcal{J} : & \quad \{\neg \text{at}_0, \neg \text{at}_1, \neg \text{at}_m\}.
\end{align*}
\]

\[
\begin{align*}
\mathcal{C} : & \quad \emptyset.
\end{align*}
\]
Let $\mathcal{D}$ be an FDS for which the above components have been identified. The state $s'$ is defined to be a $\mathcal{D}$-successor of state $s$ if

$$\langle s, s' \rangle \models \rho_{\mathcal{D}}(V, V').$$

We define a computation of $\mathcal{D}$ to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, \ldots,$$

satisfying the following requirements:

- **Initiality:** $s_0$ is initial, i.e., $s_0 \models \Theta$.
- **Consecution:** For each $j = 0, 1, \ldots$, the state $s_{j+1}$ is a $\mathcal{D}$-successor of the state $s_j$.
- **Justice:** For each $J \in \mathcal{J}$, $\sigma$ contains infinitely many $J$-positions
- **Compassion:** For each $\langle p, q \rangle \in \mathcal{C}$, if $\sigma$ contains infinitely many $p$-positions, it must also contain infinitely many $q$-positions.
Examples of Computations

Identification of the FDS $\mathcal{D}_P$ corresponding to a program $P$ gives rise to a set of computations $\text{Comp}(P) = \text{Comp}(\mathcal{D}_P)$.

The following computation of program ANY-Y corresponds to the case that $m_0$ is the first executed statement:

\[
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\ell_0} \\
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\tau_I} \ldots \xrightarrow{\tau_I} \ldots
\]

The following computation corresponds to the case that statement $\ell_1$ is executed before $m_0$.

\[
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi_1: \ell_1, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 1 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\ell_0} \\
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\tau_I} \ldots \xrightarrow{\tau_I} \ldots
\]

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement $\ell_0$ $n$ times and then terminates in the final state

\[
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: n \rangle.
\]
A Non-Computation

While we can delay termination of the program for an arbitrary long time, we cannot postpone it forever.

Thus, the sequence

\[
\langle \pi_1: l_0, \pi_2: m_0 ; x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0 ; x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0 ; x: 0, y: 1 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0 ; x: 0, y: 1 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0 ; x: 0, y: 2 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0 ; x: 0, y: 2 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0 ; x: 0, y: 3 \rangle \xrightarrow{\ell_0} \ldots
\]

in which statement \( m_0 \) is never executed is not an admissible computation. This is because it violates the justice requirement \( \neg at_{\text{m}_0} \) contributed by statement \( m_0 \), by having no states in which this requirement holds.

This illustrates how the requirement of justice ensures that program ANY-Y always terminates.

Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving.
Justice is not Enough. You also Need Compassion

The following program MUX-SEM, implements mutual exclusion by semaphores.

\[ y : \text{natural initially } y = 1 \]

\[
\begin{align*}
\ell_0 &: \text{ loop forever do} \\
\ell_1 &: \text{Non-critical} \\
\ell_2 &: \text{request } y \\
\ell_3 &: \text{Critical} \\
\ell_4 &: \text{release } y \\
\end{align*}
\]

\[
\begin{align*}
m_0 &: \text{ loop forever do} \\
m_1 &: \text{Non-critical} \\
m_2 &: \text{request } y \\
m_3 &: \text{Critical} \\
m_4 &: \text{release } y \\
\end{align*}
\]

\[ P_1 \quad - \quad P_2 \]

The semaphore instructions request \( y \) and release \( y \) respectively stand for

\[ \langle \text{await } y > 0 ; \ y := y - 1 \rangle \quad \text{and} \quad y := y + 1. \]

The compassion set of this program consists of

\[ C : \{ (\text{at}_\ell_2 \land y > 0, \text{at}_\ell_3), \ (\text{at}_m_2 \land y > 0, \text{at}_m_3) \}. \]
Program MUX-SEM

should satisfy the following two requirements:

- **Mutual Exclusion** – No computation of the program can include a state in which process $P_1$ is at $\ell_3$ while $P_2$ is at $m_3$.

- **Accessibility** – Whenever process $P_1$ is at $\ell_2$, it shall eventually reach its critical section at $\ell_3$. Similar requirement for $P_2$.

Consider the state sequence:

\[
\sigma: \langle \ell_0, m_0, 1 \rangle \rightarrow \cdots \rightarrow \langle \ell_2, m_2, 1 \rangle \xrightarrow{m_2} \langle \ell_2, m_2, 1 \rangle \xrightarrow{m_2} \langle \ell_2, m_2, 1 \rangle \xrightarrow{m_2} \langle \ell_2, m_2, 1 \rangle \xrightarrow{m_2} \langle \ell_2, m_2, 1 \rangle
\]

which violates **accessibility** for process $P_1$. Due to the requirement of **compassion** for $\ell_2$, it is not a computation, and accessibility is guaranteed.

**Conclusion:** Justice alone is not sufficient !!!
SPL: Syntax

In the following, let $b$ be a boolean expression, $r$ be a natural variable, and $S, S_1, \ldots, S_k$ be statements.

- For a variable $y$ and an expression $e$ of compatible type, $y := e$ is an assignment statement.

- `await $b$` is an await statement. It awaits for $b$ to become true, and then terminates.

- `request $r$` is a request statement. It is enabled only when $r > 0$ and, when executed, it decrements $r$ by 1.

- `release $r$` is a release statement. It increments $r$ by 1.

- **Critical** and **Non-critical** are schematic statements. They are used to denote sections in mutual-exclusion programs.
Compound Statements

- **if** $b$ **then** $S_1$ **else** $S_2$ **is a conditional statement.** If $b$ is true, execution proceeds to $S_1$, otherwise to $S_2$.

- $S_1; S_2; \cdots; S_k$ **is a concatenation statement.** It executes $S_1, \ldots, S_k$ sequentially.

- $S_1$ **or** $S_2$ **or** $\cdots$ **or** $S_k$ **is a selection statement.** It non-deterministically chooses an enabled statement among $S_1, \ldots, S_k$ and proceeds to execute it.

- **while** $b$ **do** $S$ **is a while statement.** Statement $S$ is repeatedly executed as long as $b$ holds.
Communication Statements

There are two communication statements:

- $\alpha \leftarrow e$ is a send statement. It sends the value of expression $e$ onto channel $\alpha$.

- $\alpha \Rightarrow x$ is a receive statement. It reads a message from channel $\alpha$ (if non-empty) and places it in variable $x$.

There are 3 kinds of communication modes. They are distinguished by the declaration of the channel along which the message is transferred:

- $\alpha : \text{channel}[1..] \text{ of } \tau$ — declares an asynchronous channel with unbounded buffering capacity which can transmit messages (values) of type $\tau$.

- $\alpha : \text{channel}[1..k] \text{ of } \tau$ — declares an asynchronous channel with $k$-bounded buffering capacity which can transmit messages (values) of type $\tau$.

- $\alpha : \text{channel of } \tau$ — declares a synchronous channel which can transmit one message of type $\tau$ at a time.
Programs

A program $P$ has the form

\[
\text{declaration;} \; P_1 \| \cdots \| P_m
\]

where each $P_i$ is a process having the form

\[
[\text{declaration;} \; \text{statement}]
\]

Programs and processes may optionally be named.

A declaration consists of a sequence of declaration statements of the form

\[
\text{variable, \ldots, variable: type where } \varphi
\]

Each declaration statement lists several variables that share a common type and identifies their type. We use basic types such as integer, character, etc., as well as structured types, such as array, list, and set. The optional assertion $\varphi$ imposes constraints on the initial values of the variables declared in this statement.

Let $\varphi_1, \ldots, \varphi_n$ be the assertions appearing in the declaration statements of a program. We refer to the conjunction

$\varphi: \varphi_1 \land \cdots \land \varphi_n$ as the data-precondition of the program.
SPL: Semantics

Let $P :: declaration; P_1 \parallel \cdots \parallel P_m$ be a program. We proceed to construct the FDS $D_P$ corresponding to program $P$.

Let $L_i$ denote the set of locations within process $P_i$, $i = 1, \ldots, k$.

State Variables

The state variables $V$ for system $D_P$ consist of the data variables $Y$ which are declared at the head of the program and its processes, and the control variables $\Pi = \{\pi_1, \ldots, \pi_m\}$, one for each process. The data variables $Y$ range over their respectively declared data domains. The control variable (program counter) $\pi_i$ ranges over the location set of $L_i$, for $i = 1, \ldots, m$. The value of $\pi_i$ in a state denotes the current location of control in the execution of process $P_i$.

For each declared channel $\alpha$ of type $\tau$, we define variable $\alpha$ whose type is list of $\tau$.

For given locations $\ell_j, \ell_k \in L_i$, we write $at_{-\ell_j}$ as an abbreviation for $\pi_i = \ell_j$ and write $at'_{-\ell_k}$ as an abbreviation for $\pi_i' = \ell_k$. 
The Initial Condition

Let $\varphi$ denote the data precondition of program $P$. We define the initial condition $\Theta$ for $\mathcal{D}_P$ as

$$\Theta: \quad \pi_1 = \ell_1^0 \land \cdots \land \pi_m = \ell_m^0 \land \varphi,$$

where, $\ell_i^0$ is the initial location of process $P_i$. This implies that the first state in an execution of the program has the control variables pointing to the initial locations of the processes, and the data variables satisfying the data precondition.

For each channel $\alpha$, $\varphi$ includes the conjunct $\alpha = \Lambda$, where $\Lambda$ denotes the empty list.
Transition Relation, Justice, and Compassion

For each type of statement, we indicate the disjunct contributed to the transition relation, the justice, and the compassion requirements contributed by the statement. We denote by \( P_i \) the process to which the considered statement belongs.

We use the notation \( \text{pres}(U) \) as an abbreviation for

\[
\text{pres}(U) : \bigwedge_{y \in U} (y' = y),
\]

stating that all the variables in the variable set \( U \subseteq V \) are preserved by the considered statement.

- The assignment statement \( \ell_j : y := e; \ell_k : \) contributes to \( \rho \) the disjunct

\[
\text{at}_\ell_j \land \text{at}'_{\ell_k} \land y' = e \land \text{pres}(V - \{\pi_i, y\})
\]

and contributes to \( J \) the requirement \( \neg \text{at}_\ell_j \).
- The `await` statement \( \ell_j : \text{await } b; \ell_k \) : contributes to \( \rho \) the disjunct

\[
\text{at}_j \land \text{at}'_k \land b \land \text{pres}(V - \{\pi_i\})
\]

and contributes to \( \mathcal{J} \) the requirement \( \neg(\text{at}_j \land b) \), disallowing an execution which stays forever at \( \ell_j \) while \( b \) continuously holds.

- The `request` statement \( \ell_j : \text{request } r; \ell_k \) : contributes to \( \rho \) the disjunct

\[
\text{at}_j \land \text{at}'_k \land r > 0 \land r' = r - 1 \land \text{pres}(V - \{\pi_i, r\})
\]

and contributes to \( \mathcal{C} \) the requirement \( (\text{at}_j \land r > 0, \neg\text{at}_j) \).

- The `release` statement \( \ell_j : \text{release } r; \ell_k \) : contributes to \( \rho \) the disjunct

\[
\text{at}_j \land \text{at}'_k \land r' = r + 1 \land \text{pres}(V - \{\pi_i, r\})
\]

and contributes to \( \mathcal{J} \) the requirement \( \neg\text{at}_j \).
- The statement $\ell_j : \text{Non-Critical}; \ell_k :$ contributes to $\rho$ the disjunct

$$at_\ell \ell_j \land at'_\ell \ell_k \land \text{pres}(V - \{\pi_i\})$$

and does not contribute any fairness requirement. This corresponds to the assumption that non-critical sections may fail to terminate.

- The statement $\ell_j : \text{Critical}; \ell_k :$ contributes to $\rho$ the disjunct

$$at_\ell \ell_j \land at'_\ell \ell_k \land \text{pres}(V - \{\pi_i\})$$

and contributes to $\mathcal{J}$ the requirement $\neg at_\ell \ell_j$. In contrast to non-critical sections, critical sections must terminate.
Compound Statements

- The **conditional** statement $\ell_j : \text{if } b \text{ then } \ell_1 : S_1 \text{ else } \ell_2 : S_2$ contributes to $\rho$ the disjunct

$$\text{at}_{-\ell_j} \land \left( \begin{array}{c} b \land \text{at'}_{-\ell_1} \\ \lor \\ \neg b \land \text{at'}_{-\ell_2} \end{array} \right) \land \text{pres}(V - \{\pi_i\})$$

and contributes to $J$ the requirement $\neg\text{at}_{-\ell_j}$.

- The **while** statement $\ell_j : \text{while } b \text{ do } [\ell_1 : S_1]; \ell_k :$ contributes to $\rho$ the disjunct

$$\text{at}_{-\ell_j} \land \left( \begin{array}{c} b \land \text{at'}_{-\ell_1} \\ \lor \\ \neg b \land \text{at'}_{-\ell_k} \end{array} \right) \land \text{pres}(V - \{\pi_i\})$$

and contributes to $J$ the requirement $\neg\text{at}_{-\ell_j}$. 
Asynchronous Communication

- Let $\alpha$ be an asynchronous channel with buffering capacity $k$ which is either a positive integer or the special symbol $\infty$ for the case of unbounded buffering capacity.

The asynchronous send statement $\ell_j : \alpha \leftarrow e; \ell_k :$ contributes to $\rho$ the disjunct:

$$\text{at}_{\ell_j} \land |\alpha| < k \land \text{at}'_{\ell_k} \land \alpha' = \alpha * (e) \land \text{pres}(V - \{\pi_i, \alpha\})$$

where $|\alpha|$ is the length of the list $a$, and $\alpha * (e)$ is obtained by appending the value of $e$ to the end of the list $\alpha$. This statement also contributes to $C$ the requirement $(\text{at}_{\ell_j} \land |\alpha| < k, \neg \text{at}_{\ell_j})$. Note that the condition $|\alpha| < \infty$ is always true.

- The receive statement $\ell_j : \alpha \Rightarrow x; \ell_k :$, where $\alpha$ is an asynchronous channel, contributes to $\rho$ the disjunct

$$\text{at}_{\ell_j} \land \alpha \neq \Lambda \land \text{at}'_{\ell_k} \land x' = \text{hd}(\alpha) \land \alpha' = \text{tl}(\alpha) \land \text{pres}(V - \{\pi_i, x, \alpha\})$$

where $\text{hd}$ and $\text{tl}$ are the head and tail list operations, respectively. It also contributes to $C$ the compassion requirement $(\text{at}_{\ell_j} \land \alpha \neq \Lambda, \neg \text{at}_{\ell_j})$. 

Synchronous Communication

Let $\alpha$ be a synchronous channel. Each pair of matching send and receive statements:

\[ \ell_j : \alpha \leftarrow e; \; \ell_k : \quad \text{and} \quad m_u : \alpha \Rightarrow x; \; m_v : \]

contributes to $\rho$ the disjunct:

\[ \text{at}_{-\ell_j} \land \text{at}_{-\ell_k} \land \text{at}_{-m_u} \land \text{at}_{-m_v} \land x' = e \land \text{pres}(V - \{x, \pi(\ell_j), \pi(m_u)\}) \]

where $\pi(\ell_j)$ (similarly $\pi(m_u)$) is the program counter $\pi_i$ ($\pi_r$) for process $P_i$ ($P_r$) such that $\ell_j \in L_i$ ($m_u \in L_r$).

Such a pair also contributes to $C$ the requirement

\[ (\text{at}_{-\ell_j} \land \text{at}_{-m_u}, \; \text{at}_{-\ell_k} \land \text{at}_{-m_v}) \]
The Idling Transition

In addition to the above, the transition relation always contains the disjunct

$$\rho_I : \pres(V)$$

Note that the concatenation and selection statements do not contribute any disjuncts of their own to $$\rho$$. Any action performed by one of these statements can be attributed to one of their sub-statements.
Requirement Specification Language: Linear Temporal Logic

Assume an underlying (first-order) assertion language. The predicate $at_{\ell_i}$ abbreviates the formula $\pi_j = \ell_i$, where $\ell_i$ is a location within process $P_j$.

A temporal formula is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\lor$ and the basic temporal operators:

- $\Diamond$ – Next
- $\Box$ – Previous
- $U$ – Until
- $S$ – Since

Other temporal operators can be defined in terms of the basic ones as follows:

- $\Diamond p = 1 U p$ – Eventually
- $\Box p = \neg \Diamond \neg p$ – Henceforth
- $p W q = \Box p \lor (p U q)$ – Waiting-for, Unless, Weak Until
- $\Diamond p = 1 S p$ – Sometimes in the past
- $\Box p = \neg \Diamond \neg p$ – Always in the past
- $p B q = \Box p \lor (p S q)$ – Back-to, Weak Since

A model for a temporal formula $p$ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state $s_j$ provides an interpretation for the variables of $p$. 
Semantics of LTL

Given a model $\sigma$, we define the notion of a temporal formula $p$ holding at a position $j \geq 0$ in $\sigma$, denoted by $(\sigma, j) \models p$:

- For an assertion $p$,
  
  $$(\sigma, j) \models p \iff s_j \models p$$

  That is, we evaluate $p$ locally on state $s_j$.

- $$(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$$

- $$(\sigma, j) \models p \lor q \iff (\sigma, j) \models p \text{ or } (\sigma, j) \models q$$

- $$(\sigma, j) \models \Box p \iff (\sigma, j + 1) \models p$$

- $$(\sigma, j) \models p \mathcal{U} q \iff \text{for some } k \geq j, (\sigma, k) \models q, \text{ and for every } i \text{ such that } j \leq i < k, (\sigma, i) \models p$$

- $$(\sigma, j) \models \Diamond p \iff j > 0 \text{ and } (\sigma, j - 1) \models p$$

- $$(\sigma, j) \models p \mathcal{S} q \iff \text{for some } k \leq j, (\sigma, k) \models q, \text{ and for every } i \text{ such that } j \geq i > k, (\sigma, i) \models p$$

This implies the following semantics for the derived operators:

- $$(\sigma, j) \models \square p \iff (\sigma, k) \models p \text{ for all } k \geq j$$

- $$(\sigma, j) \models \Diamond p \iff (\sigma, k) \models p \text{ for some } k \geq j$$
If \((\sigma, 0) \models p\) we say that \(p\) holds over \(\sigma\) and write \(\sigma \models p\). Formula \(p\) is **satisfiable** if it holds over some model. Formula \(p\) is **(temporally) valid** if it holds over all models.

Formulas \(p\) and \(q\) are **equivalent**, denoted \(p \sim q\), if \(p \leftrightarrow q\) is valid. They are called **congruent**, denoted \(p \approx q\), if \(\Box (p \leftrightarrow q)\) is valid. If \(p \approx q\) then \(p\) can be replaced by \(q\) in any context.

The **entailment** \(p \Rightarrow q\) is an abbreviation for \(\Box (p \rightarrow q)\).
Reading Exercises

Following are some temporal formulas $\varphi$ and what do they say about a sequence $\sigma : s_0, s_1, \ldots$ such that $\sigma \models \varphi$:

- $p \rightarrow \lozenge q$ — If $p$ holds at $s_0$, then $q$ holds at $s_j$ for some $j \geq 0$.

- $\Box (p \rightarrow \lozenge q)$ — Every $p$ is followed by a $q$. Can also be written as $p \Rightarrow \lozenge q$.

- $\Box \lozenge q$ — The sequence $\sigma$ contains infinitely many $q$’s.

- $\Box \Box q$ — All but finitely many states in $\sigma$ satisfy $q$. Property $q$ eventually stabilizes.

- $q \Rightarrow \lozenge p$ — Every $q$ is preceded by a $p$ — causality.

- $(\neg r) W q$ — $q$ precedes $r$. $r$ cannot occur before $q$ — precedence. Note that $q$ is not guaranteed, but $r$ cannot happen without a preceding $q$.

- $(\neg r) W (q \land \neg r)$ — $q$ strongly precedes $r$.

- $p \Rightarrow (\neg r) W q$ — Following every $p$, $q$ precedes $r$. 
**Temporal Specification of Properties**

Formula $\varphi$ is *$D$-valid*, denoted $D \models \varphi$, if all computations of $D$ satisfy $\varphi$. Such a formula specifies a property of $D$.

Following is a *temporal* specification of the main properties of program MUX-SEM.

- **Mutual Exclusion** – No computation of the program can include a state in which process $P_1$ is at $l_3$ while $P_2$ is at $m_3$. Specifiable by the formula

  $$\Box \neg (at_{-l_3} \land at_{-m_3})$$

- **Accessibility** for $P_1$ – Whenever process $P_1$ is at $l_2$, it shall eventually reach its critical section at $l_3$. Specifiable by the formula

  $$\Box (at_{-l_2} \rightarrow \Diamond at_{-l_3})$$
Expressive Completeness

Every (propositional) temporal formula $\varphi$ can be translated into a first-order logic with monadic predicates over the naturals ordered by $<$ (1st-order theory of linear order).

For example, the 1st-order translation of $p \Rightarrow \Diamond q$ is

$$\forall t_1 \geq 0 : (p(t_1) \rightarrow \exists t_2 \geq t_1 : (q(t_2)))$$

Can every 1st-order formula be translated into temporal logic?

W. Kamp [Kamp68] has shown that the answer is negative if we only allow $\Box$ and $\Diamond$ in our temporal formulas. But then proceeded to show that:

**Claim 1.** Every 1st-order formula can be translated into a temporal formula in the logic $L(\O, \U, \Box, S)$.

[GPSS81] has shown that

**Claim 2.** Every 1st-order formula can be translated into a temporal formula in the logic $L(\O, \U)$.

This also shows that the past operators add no expressive power.
Classification of Formulas/Properties

A formula of the form $\square p$ for some past formula $p$ is called a safety formula.

A formula of the form $\square \Diamond p$ for some past formula $p$ is called a response formula.

An equivalent characterization is the form $p \Rightarrow \Diamond q$. The equivalence is justified by

$$\square (p \rightarrow \Diamond q) \sim \square \Diamond ((\neg p) \mathcal{B} q)$$

Both formulas state that either there are infinitely many $q$’s, or there there are no $p$’s, or there is a last $q$-position, beyond which there are no further $p$’s.

A property is classified as a safety/response property if it can be specified by a safety/response formula.

Every temporal formula is equivalent to a conjunction of a reactivity formulas, i.e.

$$\bigwedge_{i=1}^{k} (\square \Diamond p_i \lor \Diamond \square q_i)$$