Modeling and Analysis of Real-Time and Hybrid Systems

Amir Pnueli

Tuesdays, 5-7 PM

Copies of presentations and Lecture Notes will be available at
http://www.cs.nyu.edu/courses/fall03/G22.3033-007/index.htm

Course Contents

- Introduce formal models for programs (designs) and their specification.
- Study the Verification problem: Given a program P and a specification φ, establish that P satisfies φ.
- Study the Synthesis problem: Given a specification φ, construct a program P which satisfies φ (if one exists).
- Repeat these for the model classes of Reactive, Real-Time, and Hybrid Systems.

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

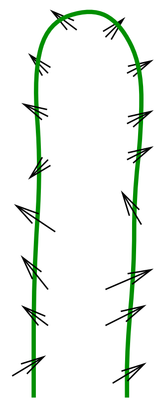
$$y = x^2.$$

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 linear temporal logic (LTL).
- An Implementation Language for describing proposed implementations (both software and hardware). Use SPL, a simple programming language.
- Verification Techniques for validating that an implementation satisfies a specification. Practiced approaches:
  - Algorithmic verification methods for exploratory verification of finite-state systems: Enumerative and Symbolic variants.
  - A deductive methodology based on theorem-proving methods. Can accommodate infinite-state systems, but requires user interaction.

A Hierarchy of Computational Models

Reactive Systems

Expresses precedence relations between events

↓

Real Time

Can measure temporal distance

$$\square\,(request \longrightarrow \diamondsuit_{\leq 5}\, response)$$

↓

Hybrid Systems

Combination of discrete and Continuous Components

Why is Real-Time Intermediate between Reactive and Hybrid Systems?

Hybrid systems allow multiple continuous variables whose rate of change is unsynchronized.

Real time systems allow a single continuous variable — time. Alternate models allow several clocks, but their rate of change is fully synchronized.

Hard and Soft Real Time:
The techniques introduced in this course enable the modeling and analysis of real-time systems which, usually involve several interacting components. This is sometimes described as hard real time, because any failure to meet a deadline is considered catastrophic,

Another school of techniques concentrates on the reliable construction of a real-time system implemented on a single computer which uses time slicing to serve multiple environment agents. There, a strong emphasis is put on scheduling strategies.

Recently, we have witnessed a new class of real-time applications, where some failure to meet a deadline is tolerable, as long as the system provides an acceptable quality of service.

The latter two cases are often grouped under the name soft real time.

Fair Discrete Systems

A fair discrete system (FDS) $\mathcal{D} = \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ consists of:

- V – A finite set of typed state variables. A V-state s is an interpretation of V. Σ_V – the set of all V-states.
- $\mathcal{O} \subseteq V$ – A set of observable variables.
- Θ – An initial condition. A satisfiable assertion that characterizes the initial states.
- ρ – 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 $x := x + 1$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$ A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many J_i-states for each J_i, $i = 1, \ldots, k$.
- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$ 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

$$x,\ y : \textbf{natural initially } x = y = 0$$

$$\left[\begin{array}{l} \ell_0 : \quad \textbf{while } x = 0 \textbf{ do} \\ \qquad [\ell_1 : \ y := y + 1] \\ \ell_2 : \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \quad x := 1 \\ m_1 : \end{array}\right]$$

$$- \quad P_1 \quad - \qquad\qquad\qquad - \quad P_2 \quad -$$

The Corresponding FDS

- State Variables V: $\begin{pmatrix} x, y & : & \text{natural} \\ \pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 & : & \{m_0, m_1\} \end{pmatrix}$.

- Initial condition:

$$\Theta: \ \pi_1 = \ell_0 \ \wedge \ \pi_2 = m_0 \ \wedge \ x = y = 0.$$

- Transition Relation: ρ: $\rho_I \vee \rho_{\ell_0} \vee \rho_{\ell_1} \vee \rho_{m_0}$, with appropriate disjunct for each statement. For example, the disjuncts ρ_I and ρ_{ℓ_0} are

$$\rho_I: \quad \pi_1' = \pi_1 \wedge \pi_2' = \pi_2 \wedge x' = x \wedge y' = y$$

$$\rho_{\ell_0}: \quad \pi_1 = \ell_0 \quad \wedge \quad \begin{pmatrix} x = 0 \ \wedge \ \pi_1' = \ell_1 \\ \vee \\ x \neq 0 \ \wedge \ \pi_1' = \ell_2 \end{pmatrix}$$

$$\wedge \quad \pi_2' = \pi_2 \wedge x' = x \wedge y' = y$$

- Justice set: $\mathcal{J}$: $\{\neg at\_\ell_0, \neg at\_\ell_1, \neg at\_m_0\}$.

- Compassion set: $\mathcal{C}$: $\emptyset$.

Computations

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

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

satisfying the following requirements:

- Initiality: s_0 is initial, i.e., $s_0 \models \Theta$.
- Consecution: For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.
- Justice: For each $J \in \mathcal{J}$, σ contains infinitely many J-positions
- Compassion: For each $\langle p, q \rangle \in \mathcal{C}$, if σ 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 $\mathcal{C}omp(P) = \mathcal{C}omp(\mathcal{D}_P)$.

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

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

The following computation corresponds to the case that statement ℓ_1 is executed before m_0.

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

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement ℓ_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

$$\langle \pi_1\colon \ell_0 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 0\rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1\colon \ell_0 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 1\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 1\rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1\colon \ell_0 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 2\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 2\rangle \xrightarrow{\ell_1}$$
$$\langle \pi_1\colon \ell_0 \,,\, \pi_2\colon m_0 \;;\; x\colon 0 \,,\, y\colon 3\rangle \xrightarrow{\ell_0} \cdots$$

in which statement m_0 is never executed is not an admissible computation. This is because it violates the justice requirement $\neg at\_m_0$ 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.

$$y: \textbf{ natural initially } y = 1$$

$$\left[\begin{array}{ll} \ell_0: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} \ell_1: & \textbf{Non-critical} \\ \ell_2: & \textbf{request } y \\ \ell_3: & \textbf{Critical} \\ \ell_4: & \textbf{release } y \end{array}\right] \end{array}\right] \parallel \left[\begin{array}{ll} m_0: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} m_1: & \textbf{Non-critical} \\ m_2: & \textbf{request } y \\ m_3: & \textbf{Critical} \\ m_4: & \textbf{release } y \end{array}\right] \end{array}\right]$$

$$- \quad P_1 \quad - \qquad\qquad\qquad - \quad P_2 \quad -$$

The semaphore instructions **request** y and **release** y respectively stand for

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

The compassion set of this program consists of

$$\mathcal{C}: \quad \{(at\_\ell_2 \ \wedge\ y > 0,\ at\_\ell_3), \quad (at\_m_2 \ \wedge\ y > 0,\ at\_m_3)\}.$$

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 ℓ_3 while P_2 is at m_3.

- Accessibility – Whenever process P_1 is at ℓ_2, it shall eventually reach it's critical section at ℓ_3. Similar requirement for P_2.

Consider the state sequence:

$$
\begin{array}{lllll}
\sigma: & \langle \ell_0,\ m_0,\ 1\rangle \longrightarrow & \cdots \longrightarrow & & \boxed{\langle \ell_2,\ m_2,\ 1\rangle} \xrightarrow{m_2} \\
& \langle \ell_2,\ m_3,\ 0\rangle \xrightarrow{m_3} & \langle \ell_2,\ m_4,\ 0\rangle \xrightarrow{m_4} & & \\
& \langle \ell_2,\ m_0,\ 1\rangle \xrightarrow{m_0} & \langle \ell_2,\ m_1,\ 1\rangle \xrightarrow{m_1} & & \boxed{\langle \ell_2,\ m_2,\ 1\rangle} \xrightarrow{m_2} \\
& \langle \ell_2,\ m_3,\ 0\rangle \longrightarrow & \cdots\ , & &
\end{array}
$$

which violates accessibility for process P_1. Due to the requirement of compassion for ℓ_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.

- **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.

Programs

A program P has the form

$$declaration; \; P_1 \| \cdots \| P_m$$

where each P_i is a process having the form

$$[declaration; statement]$$

Programs and processes may optionally be named.

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

$variable, \; \ldots, \; variable\colon \; type$ **where** φ

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 φ 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 $P :: \quad \mathit{declaration};\ P_1 \parallel \cdots \parallel P_m$ be a program. We proceed to construct the FDS $\mathcal{D}_P$ 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 $\mathcal{D}_P$ 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 π_i ranges over the location set of L_i, for $i = 1, \ldots, m$. The value of π_i in a state denotes the current location of control in the execution of process P_i.

For given locations $\ell_j, \ell_k \in L_i$, we write at_ℓ_j as an abbreviation for $\pi_i = \ell_j$ and write at'_ℓ_k as an abbreviation for $\pi'_i = \ell_k$.

Observable Variables At this point, we take $\mathcal{O} = V$.

The Initial Condition Let φ denote the data precondition of program P. We define the initial condition Θ for $\mathcal{D}_P$ as

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

where, ℓ_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.

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 *pres*(U) as an abbreviation for

$$\textit{pres}(U): \quad \bigwedge_{y \in U} (y' = y),$$

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 ρ the disjunct

  $$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ y' = e \ \wedge \ \textit{pres}(V - \{\pi_i, y\})$$

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

- The await statement $\ell_j :$ **await** $b;\ \ell_k :$ contributes to ρ the disjunct

  $$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ b \ \wedge \ \textit{pres}(V - \{\pi_i\})$$

  and contributes to $\mathcal{J}$ the requirement $\neg(at\_\ell_j \ \wedge \ b)$, disallowing an execution which stays forever at ℓ_j while b continuously holds.

- The request statement ℓ_j : **request** r; ℓ_k : contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ r > 0 \ \wedge \ r' = r - 1 \ \wedge \ \mathit{pres}(V - \{\pi_i, r\})$$

and contributes to $\mathcal{C}$ the requirement $(at\_\ell_j \wedge r > 0, \ at\_\ell_k)$.

- The release statement ℓ_j : **release** r; ℓ_k : contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ r' = r + 1 \ \wedge \ \mathit{pres}(V - \{\pi_i, r\})$$

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

- The statement ℓ_j : **Non-Critical**; ℓ_k : contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ \mathit{pres}(V - \{\pi_i\})$$

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

- The statement ℓ_j : **Critical**; ℓ_k : contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ at'\_\ell_k \ \wedge \ \mathit{pres}(V - \{\pi_i\})$$

and contributes to $\mathcal{J}$ the requirement $\neg at\_\ell_j$. In contrast to non-critical sections, critical sections must terminate.

- The conditional statement ℓ_j : **if** b **then** ℓ_1 :S_1 **else** ℓ_2 :S_2 contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ \left(\begin{array}{ccc} b & \wedge & at'\_\ell_1 \\ & \vee & \\ \neg b & \wedge & at'\_\ell_2 \end{array}\right) \ \wedge \ pres(V - \{\pi_i\})$$

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

- The while statement ℓ_j : **while** b **do** $[\ell_1 :S_1]$; ℓ_k : contributes to ρ the disjunct

$$at\_\ell_j \ \wedge \ \left(\begin{array}{ccc} b & \wedge & at'\_\ell_1 \\ & \vee & \\ \neg b & \wedge & at'\_\ell_k \end{array}\right) \ \wedge \ pres(V - \{\pi_i\})$$

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

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

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

Note that the concatenation and selection statements do not contribute any disjuncts of their own to ρ. Any action performed by one of these statements can be attributed to one of their sub-statements.

Requirement Specification Language: Temporal Logic

Assume an underlying (first-order) assertion language. The predicate at_ℓ_i, abbreviates the formula $\pi_j = \ell_i$, where ℓ_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 $\vee$ and the basic temporal operators:

$\bigcirc$	– Next	$\ominus$	– Previous
$\mathcal{U}$	– Until	$\mathcal{S}$	– Since

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

$\diamondsuit\, p = 1\ \mathcal{U}\ p$	– Eventually
$\square\, p = \neg \diamondsuit \neg p$	– Henceforth
$p\ \mathcal{W}\ q = \square\, p \vee (p\ \mathcal{U}\ q)$	– Waiting-for, Unless, Weak Until
$\diamondminus\, p = 1\ \mathcal{S}\ p$	– Sometimes in the past
$\boxminus\, p = \neg \diamondminus \neg p$	– Always in the past
$p\ \mathcal{B}\ q = \boxminus\, p \vee (p\ \mathcal{S}\ q)$	– 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 σ, we define the notion of a temporal formula p holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models p$:

- For an assertion p,
  $(\sigma, j) \models p \quad \Longleftrightarrow \quad s_j \models p$
  That is, we evaluate p locally on state s_j.
- $(\sigma, j) \models \neg p \quad \Longleftrightarrow \quad (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \quad \Longleftrightarrow \quad (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \quad \Longleftrightarrow \quad (\sigma, j+1) \models p$
- $(\sigma, j) \models p\, \mathcal{U}\, q \quad \Longleftrightarrow \quad$ for some $k \geq j, (\sigma, k) \models q$,
  and for every i such that $j \leq i < k, \quad (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \quad \Longleftrightarrow \quad j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p\, \mathcal{S}\, q \quad \Longleftrightarrow \quad$ for some $k \leq j, (\sigma, k) \models q$,
  and for every i such that $j \geq i > k, \quad (\sigma, i) \models p$

This implies the following semantics for the derived operators:

- $(\sigma, j) \models \Box\, p \quad \Longleftrightarrow \quad (\sigma, k) \models p$ for all $k \geq j$
- $(\sigma, j) \models \Diamond\, p \quad \Longleftrightarrow \quad (\sigma, k) \models p$ for some $k \geq j$

If $(\sigma, 0) \models p$ we say that p holds over σ 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 $p \sim q$ 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 φ and what do they say about a sequence $\sigma : s_0, s_1, \ldots$ such that $\sigma \models \varphi$:

- $p \rightarrow \diamondsuit q$ — If p holds at s_0, then q holds at s_j for some $j \geq 0$.
- $\square (p \rightarrow \diamondsuit q)$ — Every p is followed by a q. Can also be written as $p \Rightarrow \diamondsuit q$.
- $\square \diamondsuit q$ — The sequence σ contains infinitely many q's.
- $\diamondsuit \square q$ — All but finitely many states in σ satisfy q. Property q eventually stabilizes.
- $q \Rightarrow \diamondminus p$ — Every q is preceded by a p — causality.
- $(\neg r) \mathcal{W} q$ — q precedes r. r cannot occur before q — precedence. Note that q is not guaranteed, but r cannot happen without a preceding q.
- $(\neg r) \mathcal{W} (q \wedge \neg r)$ — q strongly precedes r.
- $p \Rightarrow (\neg r) \mathcal{W} q$ — Following every p, q precedes r.

Temporal Specification of Properties

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

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

$$\Box\ \neg(at\_\ell_3\ \wedge\ at\_m_3)$$

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

$$\Box\ (at\_\ell_2 \quad \rightarrow \quad \Diamond\ at\_\ell_3)$$

Expressive Completeness

Every (propositional) temporal formula φ 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 $p \Rightarrow \diamondsuit\, q$ 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 $\square$ and $\diamondsuit$ 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* $\mathcal{L}(\bigcirc, \mathcal{U}, \ominus, \mathcal{S})$.

[GPSS81] has shown that

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

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 $\square \diamondsuit\, p$ for some past formula p is called a response formula.

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

$$\square\,(p \rightarrow \diamondsuit\, q) \qquad \sim \qquad \square \diamondsuit\,((\neg p)\ \mathcal{B}\ q)$$

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.

$$\bigwedge_{i=1}^{k} (\square \diamondsuit\, p_i \ \vee\ \diamondsuit \square\, q_i)$$