
Mondays, 5:00-6:50 PM

Amir Pnueli

Timed and Hybrid Systems
Course grades will be determined based on assignments and a term project.

Course Outline

- Extensions of the models and specification logics to real-time and hybrid systems.
- 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.
- Systems and their specification logics to real-time and hybrid systems.

Main topics we will consider are:

The course will focus on formal verification of reactive, real-time, and hybrid systems.
Lecture 1: Modeling Systems

Course Contents

- Introduce formal models for programs (designs) and their specification.

- Study the Verification problem: Given a program $P$ and a specification $\phi$, establish that $P$ satisfies $\phi$ (if one exists).

- Study the Synthesis problem: Given a specification $\phi$, construct a program $P$ which satisfies $\phi$ (if one exists).

- Repeat these for the model classes of Reactive, Real-Time, and Hybrid Systems.

Timed and Hybrid Systems, NYU, Spring, 2007

A. Pnueli
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 $y = 1 + 3 + (2x^1)$ can be specified by the requirement $y = x^2$.

Such programs must be specified and verified in terms of their behaviors.

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

A specificication language for specifying systems and their properties. We use linear temporal logic (LTL).

A specificication language for exploratory verification of finite-state systems. Use SRL, a simple programming language (both software and hardware). Use fair探索 and proving methods for exploring properties of finite-state systems.

A deductive methodology based on theorem-proving methods. Can accommodate infinite-state systems but requires user interaction.

A framework for reactive systems verification. We use lineartemporal logic (LTL).
A Hierarchy of Computational Models

Hybrid Systems

Combination of discrete and Continuous

Components

Real-Time

Can measure temporal distance

between events

Reactive Systems

Express precedence relations

A Pnueli
Recently, we have witnessed a new class of real-time applications, where some strategies.

There, a strong emphasis is put on scheduling multiple environment agents. There, a strong emphasis is put on scheduling multiple environment agents. The techniques introduced in this chapter 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 time systems implemented on a single computer which uses time slicing to serve several environment agents. There, a strong emphasis is put on scheduling multiple environment agents. The techniques introduced in this chapter enable the modeling and analysis of hard and soft real-time systems.

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.

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

Why is Real-Time Intermediate between Reactive and Hybrid Systems?
The latter two cases are often grouped under the name soft real time. Failure to meet a deadline is tolerable, as long as the system provides an acceptable quality of service.
A fair discrete system (FDS) consists of:

\[
\langle \mathcal{C}, \mathcal{L}, \Theta, \mathcal{O}, \Lambda \rangle = \mathcal{A}
\]
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:

\[
0 = y = x \text{ initially}
\]

\[
\text{while } 0 = x \text{ do}
\]

\[
0 = y + 1
\]

\[
\text{end while}
\]

\[
\text{end program}
\]
The Corresponding FDS

State Variables $V$: $\mu$, $x$, $y$

Initial condition: $\emptyset$: $C : \{ 0 \} : J$

Transition Relation: For example, the disjuncts and are

Initial condition: $\emptyset$: $\Theta : \emptyset$

State Variables $\land$
Computations

Let $D$ be a FDS for which the above components have been identified. The states $s_0$ is defined to be a $D$-successor of state $s$ if

\[(s, \Lambda)s = s_{i+1}\]

We define a computation of $D$ to be an infinite sequence of states $\langle s_0, s_1, s_2, \ldots \rangle$ satisfying the following requirements:

- **Initiality:** $s_0$ is initial, i.e., $s_0 \in \Theta$.
- **Justice:** For each $f \in F$, $s$ contains infinitely many $f$-positions.
- **Consecution:** For each $j = 0, 1, \ldots$, the state $s_{j+1}$ is a $D$-successor of the state $s_j$.
- **Compassion:** For each $h, p \in C$, if $p$ contains infinitely many $h$-positions, it must also contain infinitely many $p$-positions.
- **Timed and Hybrid Systems, NYU, Spring, 2007**
Examples of Computations

Identification of the FDS corresponding to a program gives rise to a set of computations of the FDS corresponding to a program $\mathcal{D}$. 

In a similar way, we can construct for each $0 < n$ a computation that executes

The following computation corresponds to the case that statement $I^1$ is executed before $m_0$.

The following computation corresponds to the case that statement $I^0$ is the first executed statement:

\[
\langle n : \hat{h} , I : x : \nu , \nu , \nu , \nu , \nu , \nu \rangle
\]

The body of statement $I^0$ executes $n$ times and then terminates in the final state

In a similar way, we can construct for each $n$ a computation that executes

The following computation corresponds to the case that statement $I^1$ is executed after statement $I^0$.
Thus, the sequence cannot postpone it forever.

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

Thus, the requirement of justice ensures that program ANY-Y always terminates.

This illustrates how the requirement of justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving.

Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving.

A Non-Computation
Justice is not enough. You also need compassion.

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

\[ \{ \{ y < 0 \land y < 0 \}, \{ \text{at-c}_{0}, \text{at-c}_{3} \} \} \]

The compassion set of this program consists of

\[ \{ y \land y = 0 \} \quad \text{and} \quad \{ y \land y < 0 \} \]

The semaphore instructions request_y and release_y respectively stand for

- $p_2$
- $p_1$

\[
\begin{bmatrix}
\text{Release} & : m_4 \\
\text{Critical} & : m_3 \\
\text{Request} & : m_2 \\
\text{Non-critical} & : m_1 \\
\text{Loop forever do} & : m_0
\end{bmatrix}
\|
\begin{bmatrix}
\text{Release} & : j_4 \\
\text{Critical} & : j_3 \\
\text{Request} & : j_2 \\
\text{Non-critical} & : j_1 \\
\text{Loop forever do} & : j_0
\end{bmatrix}
\]

$\hat{y} = \hat{y}^0$ initially natural initially $\hat{y}$
Lecture 1: Modeling Systems

Program MUX-SEM

A. Pnueli

Timed and Hybrid Systems, NYU, Spring 2007

Conclusion: Justice alone is not sufficient. For $P_2$, it is not a computation, and accessibility is guaranteed. Which violates accessibility for process $P_1$. Due to the requirement of compassion for $P_2$, accessibility for process $P_1$ is violated. Due to the requirement of compassion for $P_2$, it is not a computation, and accessibility is guaranteed.

Consider the state sequence:

Critical section at $P_2$. Similar requirement for $P_3$. Whenever process $P_1$ is at $P_2$, it shall eventually reach its

- Accessibility

- Mutual Exclusion

should satisfy the following two requirements:

Program MUX-SEM
section in mutual-exclusion programs.

• Critical and non-critical are schematic statements. They are used to denote

• For a variable y and an expression e of compatible type,

• For a variable y and an expression e of compatible type,

• In the following, let q be a boolean expression, t be a natural variable, and

Syntax: SPiL
Compound Statements

- \( \text{if } b \text{ then } S_1 \text{ else } S_2 \): A conditional statement. If \( b \) is true, execution proceeds to \( S_1 \), otherwise to \( S_2 \).

- \( S_1 ; S_2 ; \ldots ; S_k \): A concatenation statement. It executes \( S_1 \); \( S_2 \); \ldots ; \( S_k \) sequentially.

- \( \text{while } b \text{ do } S \): A while statement. Statement \( S \) is repeatedly executed as long as \( q \) holds.

- \( S_1 \text{ or } S_2 \text{ or } \ldots \text{ or } S_k \): A selection statement. It non-deterministically chooses an enabled statement among \( S_1 \); \( S_2 \); \ldots ; \( S_k \) and proceeds to execute it.

- \( \text{if } q \text{ then } S_1 \text{ else } S_2 \): A conditional statement. If \( q \) is true, execution proceeds to \( S_1 \), otherwise to \( S_2 \).
One message of type \( \tau \) at a time:

- \( \alpha : \text{channel of } \tau \) — declares a synchronous channel which can transmit
  boudned buffering capacity messages (values) of \( \tau \).

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

- \( \alpha : \text{channel} \) — declares an asynchronous channel with unbounded
  declaration of the channel along which the message is transmitted:

There are 3 kinds of communication modes. They are distinguished by the

empty) and places it in variable \( x \).

- \( \alpha \leftarrow x \) is a receive statement. It reads a message from channel \( \alpha \) (if non-

- \( \alpha \rightarrow e \) is a send statement. It sends the value of expression \( e \) onto channel \( \alpha \).

There are two communication statements:

**Communication Statements**
Programs and processes may optionally be named.

\[
\text{declaration; statement}
\]

where each process has the form

\[
\text{declaration; } P_1 \parallel \cdots \parallel P_m
\]
Let $P::\text{declaration}$;

Let $P_1$ be a program. We proceed to construct the FDS corresponding to program $P$.

Let $P::\text{declaration};P_1\parallel\ldots\parallel P_m$ be a program. We proceed to construct the

**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 $=\{1,\ldots,m\}$, one for each process. The control variables range over their respective declared data domains. The control variable (program counter) $i$ ranges over the location set of $P_i$, for $i = 1,\ldots,m$. The value of $i$ in a state represents the current location of control in the execution of process $P_i$.

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

For given locations $j_1,j_2\in L_i$, we write $\pi_{j_1,j_2}$ as an abbreviation for $\pi_{j_1}=\pi_{j_2}$ as an abbreviation for $\pi_j$ and $\pi_0$.

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

**List of**

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

Let $I\subseteq I^P$ denote the set of locations within process $P$.

Let $P::\text{declaration};P_1\parallel\ldots\parallel P_m$ be a program. We proceed to construct the

**SPL: Semantics**

A Pnueli Lecture 1: Modeling Systems
Let \( \theta \) denote the data precondition of program \( P \). We define the initial condition

\[
\theta \lor \omega_0 = \omega_1 \lor \cdots \lor \omega_I = \omega_V : \Theta
\]

for \( \Theta \) as follows:

For each channel \( i \), \( \alpha \) includes the conjunct \( \phi \lor \alpha \), where \( \alpha \) denotes the empty list.

The initial condition for \( P \) 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.
and contributes to the requirement -

\[ (\{y_i \} - \Lambda) \check{\text{pres}} \vee \forall e = \check{\text{pres}} \vee \forall y \check{\text{at}} \forall f \check{\text{at}} \forall f \check{\text{pre}} \]

\[ \text{The assignment statement } \]

\[ j : y := e; \]

\[ k : \text{ contributes to the disjunct } d \]

\[ \text{stating that all the variables in the variable set are preserved by the considered statement.} \]

\[ \Lambda \subseteq \Omega \]

\[ \\exists y \check{\text{pres}} (\Omega) \check{\text{pres}} \]

We use the notation \( \check{\text{pres}} (\Omega) \) as an abbreviation for belonging.

\[ \text{Transition Relation, Justice, and Compassion} \]

For each type of statement, we indicate the disjunct contributed to the transition relation, the justice, and the compassion requirements contributed by the considered statement.

\[ \text{A. Pnueli} \]

\[ \text{Timed and Hybrid Systems, NYU, Spring, 2007} \]
The await statement `j: await b;` contributes to the disjunct \((\{\{j\} - \Lambda\}) \text{ Pres} \land I + j = i \lor \exists f \text{ at } j \lor \exists f \text{ at } k\). The release statement \(j: \text{ release } r\) contributes to the disjunct \((\{\{j\} - \Lambda\}) \text{ Pres} \land I - j = i \land 0 < r \lor \exists f \text{ at } j \lor \exists f \text{ at } k\). The request statement \(j: \text{ request } r\) contributes to the disjunct \((\{\{j\} - \Lambda\}) \text{ Pres} \land q \land \exists f \text{ at } j \lor \exists f \text{ at } k\). The await statement \(j: \text{ await } q\) contributes to the disjunct \((\{\{j\} - \Lambda\}) \text{ Pres} \land q \lor \exists f \text{ at } j \lor \exists f \text{ at } k\).
Sectional, critical sections must terminate. In contrast to non-critical and contributes to the disjunct at
\[
\{\forall i - \Lambda\} \text{ press} \lor \forall i \exists j \text{ at } \{\forall j - \Lambda\} \text{ press} \lor \forall i \exists j \text{ at }
\]

The statement \( j : \text{ Critical}; k : \text{ contributesto the disjunct} \)

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

\[
\{\forall i - \Lambda\} \text{ press} \lor \forall i \exists j \text{ at } \{\forall j - \Lambda\} \text{ press} \lor \forall i \exists j \text{ at }
\]

The statement \( j : \text{ Non-Critical}; k : \text{ contributesto the disjunct} \)
Compound Statements

The conditional statement `j : if b then S1 else S2 : if q then S1 : S1 : if l then S1 : S1` contributes to the requirement \( \forall \alpha \exists \gamma \alpha \mid \neg \varphi \wedge \neg \varphi \). The while statement `j : while b do [S1 : S2 : if q then S1 : S1 : if l then S1 : S1` contributes to the requirement \( \forall \alpha \exists \gamma \alpha \mid \neg \varphi \wedge \neg \varphi \). The conditional statement \( \forall \alpha \exists \gamma \alpha \mid \neg \varphi \wedge \neg \varphi \). contributes to the requirement \( \forall \alpha \exists \gamma \alpha \mid \neg \varphi \wedge \neg \varphi \).

\( (\{\nu\} - \Lambda)pres \lor \left( \begin{array}{c} \forall \gamma \exists \nu \gamma \mid \neg \varphi \wedge \neg \varphi \\ \forall \nu \exists \gamma \nu \mid \neg \varphi \wedge \neg \varphi \end{array} \right) \lor \forall \gamma \forall \nu \exists \gamma \nu \mid \neg \varphi \wedge \neg \varphi \).
Asynchronous Communication

Let \( a \) be an asynchronous channel with buffering capacity \( k \), which is either a positive integer or the special symbol 1 for the case of unbounded buffering.

The asynchronous send statement

\[
\{\alpha, x \} - \lambda \text{press } \lor (\alpha) \mu = x \lor (\alpha) \text{hd} = x \lor (\alpha) \text{at} \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

contributes to the disjunct \( d \) in the compass requirement \( C \) where \( \mu \) and \( \nu \) are the head and tail list operations, respectively. If also

\[
(\{\alpha \} - \lambda \text{press } \lor (\alpha) \mu = x \lor (\alpha) \text{hd} = x \lor (\alpha) \text{at} \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

are the head and tail list operations, respectively. It also

\[
\text{at} \neg (\alpha) \text{at}
\]

always true.

\[
\text{at} \neg (\alpha) \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

contributes to the disjunct \( d \) in the compass requirement \( C \) where \( \alpha \) is an asynchronous channel, and \( \sigma \) is obtained by appending the value of \( e \) to the end of the list \( \alpha \). This statement also contributes to \( C \) the requirement

\[
(\{\alpha \} - \lambda \text{press } \lor (\alpha) \mu = x \lor (\alpha) \text{hd} = x \lor (\alpha) \text{at} \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

where \( |\alpha| \) is the length of the list \( \alpha \), and \( \alpha^* (\sigma) \) is obtained by appending the

condition

\[
(\{\alpha \} - \lambda \text{press } \lor (\alpha) \mu = x \lor (\alpha) \text{hd} = x \lor (\alpha) \text{at} \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

The asynchronous send statement

\[
(\{\alpha, x \} - \lambda \text{press } \lor (\alpha) \mu = x \lor (\alpha) \text{hd} = x \lor (\alpha) \text{at} \lor \forall \sigma \neq x \lor (\alpha) \text{at}
\]

contributes to the disjunct \( d \) in the compass requirement \( C \) where \( \alpha \) is an asynchronous channel with buffering capacity \( k \).
Let \(a\) be a synchronous channel. Each pair of matching send and receive statements:

**Synchronous Communication**
The Idling Transition

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

\[(\land \text{press}, \text{Id})\]
A model for a temporal formula is an infinite sequence of states $\sigma : \sigma_0, \sigma_1, \ldots$

\[
\begin{align*}
(b \land d) \land \diamond d &= b \land \Box d \\
\neg \diamond d &= \neg \Box d \\
\neg \neg \diamond d &= \Box d \\
\neg \neg \neg \diamond d &= \Diamond d \\
(b \lor d) \land \Box d &= b \lor \Box d \\
\neg \Box d &= \neg \Diamond d \\
\neg \neg \Box d &= \Diamond d \\
\neg \neg \neg \Box d &= \Diamond d
\end{align*}
\]

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

- $\neg$ Siphon
- $\lor$ Until
- $\land$ Previous
- $\lor$ Next

A temporal formula is constructed out of state formulas (assertions) to which abbreviates the formula $\forall i$, where $i$ is a location within process $P$.

Assume an underlying (first-order) assertion language. The predicate $\forall i P$,

**Requirement Specification Language: Linear Temporal Logic**
Given a model \( \phi \) of a temporal formula holding at a position \( d \) of an assertion, denoted by \( \diamond \), we define the notion of a model for a temporal formula holding at a position in a model.
If \((; 0) = p\)

we say that \(p\) holds over

\((0; 0)\).

If \((; 0) \models d\)

we say that \(d\) holds over \((0; 0)\).

\[\models (b \leftrightarrow d)\]

The entailment \(b \leftrightarrow d\) is an abbreviation for

\[\models b \leftrightarrow d\]

If \((b \leftrightarrow d)\) is valid.

Then \(b \sim d\) is called congruent, denoted (temporally) equivalent, denoted \(b \sim d\).

They are equivalent, denoted \(b \sim d\), if \((b \sim d)\) is valid.

Formulas and \(d\) are equivalent, denoted \(b \sim d\), if \((b \sim d)\) is valid.

If \((b \sim d)\) is valid.

Then \(b \leftrightarrow d\) can be replaced by \(d\) in any context.
Following every \( b \), \( d \) precedes \( b \) \( \Diamond \) \( d \) \( \leftarrow \)

\( b \) strongly precedes \( d \) \( \leftarrow \)

\( b \) does not precede \( \Diamond \) \( d \) \( \leftarrow \)

The sequence \( \Diamond \) contains infinitely many \( b \)’s.

\( b \) is followed by \( \Diamond \) \( d \) \( \leftarrow \)

\( b \) holds at \( s \) \( \Diamond \) \( d \) \( \leftarrow \)

\( \Delta \equiv \) such that \( \Diamond \) \( \rho \) \( \forall s_0, \ldots, s_l \) \( \rho \)

Following are some temporal formulas and what they say about a sequence.

Exercise Reading
Temporal Specification of Properties

A Pnueli

Mutual Exclusion – No computation of the program can include a state in which process \( P_1 \) is at \( t_3 \) while \( P_2 \) is at \( m_3 \). Specifiable by the formula:

\[(\text{at}_{t_3} \lor \text{at}_{m_3}) \square \]

Accessibility for \( P_1 \) – Whenever process \( P_1 \) is at \( t_2 \), it shall eventually reach its critical section at \( t_3 \). Specifiable by the formula at \( t_2 \)

\[\square (\text{at}_{t_3} \lor \text{at}_{m_3})\]

Timed and Hybrid Systems, NYU, Spring, 2007

Following is a temporal specification of the main properties of the program:

A Pnueli
This also shows that the past operators add no expressive power.

The logic $\mathcal{J}(\mathcal{O}, \mathcal{U}, \mathcal{S})$.

**Claim 1.** Every first-order formula can be translated into a temporal formula in

[GPSS81] has shown that

the logic $\mathcal{J}(\mathcal{O}, \mathcal{U}, \mathcal{S})$.

**Claim 2.** Every first-order formula can be translated into a temporal formula in

W. Kamp [Kamp68] has shown that the answer is negative if we only allow

in our temporal formulas. But then proceeded to show that:

Can every first-order formula be translated into temporal logic?

For example, the first-order translation of

\[ ((\exists t) b) : 1 \geq 3 \forall t \leftarrow (\exists t) d : 0 \geq 1 \forall t \]

is

\[ b \triangleleft d \]

with monadic predicates over the naturals ordered by $\geq$ (first-order theory of linear order).

Every (propositional) temporal formula \( \phi \) can be translated into a first-order logic

Expressive Completeness
Every temporal formula is equivalent to a conjunction of a reactivity formula, i.e.:

\[ (\exists b \square \Diamond p \land \exists d \Diamond \square q) \lor \gamma \]

A property is classified as a safety/response property if it can be specified by a safety/response formula. A formula of the form \( p \) for some past formula \( p \) is called a safety formula. An equivalent characterization is the formula \( \lnot (b G (d \rightarrow)) \square \lnot (b \Diamond \leftarrow d) \square \). The equivalence is justified by an equivalent characterization. A formula of the form \( d \square \) for some past formula \( d \square \) for some past formula is called a response formula. A formula of the form \( d \square \) for some past formula is called a safety formula.