Deductive Verification of Reactive Systems, NYU, Fall 2007

Wednesday, 5:00-6:50 PM

Copies of presentations and lecture notes will be available at

http://www.cs.nyu.edu/courses/fall07/G22.3033-02/index.htm

Textbooks:

• Temporal Verification of Reactive Systems: Safety by Z. Manna and A. Pnueli,

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

Deductive Verification of Reactive Systems, NYU, Fall 2007

Wednesday, 5:00-6:50 PM

Amir Pnueli
There are two classes of programs:

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

**Deductive Verification of Reactive Systems**, NYU, Fall 2007

There are two classes of programs:

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

- **Example:**
  
  The program which computes
  
  \[ y = x^2 \]

  Can be specified by the requirement
  
  \[ y = 1 + 3 + \cdots + (2x - 1) \]

  Specified in terms of Input/Output relations.

  Can be modeled as a black box.
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.

Such programs must be specified and verified in terms of their behaviors.
In this course, we concentrate on the study of deductive techniques.

A computational model providing an abstract syntactic base for all reactive systems. We use fair Discrete Systems (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). We use SPRL, a simple programming language.

Verication Techniques for validating that an implementation satisfies a specication.

Practiced approaches:
- Algorithmic verication methods for exploratory verication of nite-state systems: Enumerative and Symbolic variants.
- A deductive methodology based on theorem-proving methods. Can accommodate nite-state verication of reactive systems, but requires user interaction.
- A deductive methodology providing an abstract syntactic base for all reactive systems.
We will consider the following topics:

Course Outline

- Verifying general LTL properties.
- Verifying progress under compassion.
- Verification Diagrams.
- Rules CHAIN and WELL.
- Verifying progress properties.
- Algorithmic construction of auxiliary invariants.
- Verifying invariance properties.
- The PVS theorem prover: Encoding FDS and LTL within PVS.
- The specification language of linear temporal logic (LTL).
- A simple programming language (SPL) and its translation into FDS.
- The computational model of Fair Discrete Systems (FDS).
- Deductive Verification of Reactive Systems, NYU, Fall, 2007.
The Grand Vision

Formal verification will eventually prevail as the prime method for ensuring design correctness. Among the available techniques, deductive verification is the most powerful, and least restrictive – can prove anything.

The main obstacle hindering deductive verification from assuming its proper place are the current difficulties (both actual and conceived) in its application. These will be eventually solved by improving the available tools and proper education.

A. Pnueli
What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?

What is difficult in Deductive Verification?
A fair discrete system (FDS) consists of:

\[ \langle \mathcal{C}, \mathcal{L}, \mathcal{O}, \Theta, \mathcal{O}, \Lambda \rangle = \mathcal{A} \]
A language allowing composition of parallel processes communicating by shared variables as well as message passing.

A Simple Programming Language SPL

Example: Program ANY-Y

Consider the program

\[
\begin{align*}
-p_2
\begin{array}{l}
\left[ 
I =: x : \omega \right] \\
\left[ I + h =: h : I \right]
\end{array}
\end{align*}
\]

0 = h = x initially

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

Deductive Verification of Reactive Systems, NYU, Fall, 2007
The Corresponding FDS

\( \emptyset : \emptyset \)

Compassion set:

\( \{ 0^0 m, 1^0, 0^0, 1^0 \} : \mathcal{L} \)

Justice set:

\[ \mathcal{J} : \]

\[ \begin{align*}
\hat{\nu} = \hat{\nu} \land x = x \land \overline{\nu} = \overline{\nu} \land \\
\left( \overline{\eta} = \overline{\mu} \land 0 \neq x \right) \land \\
\left( \overline{\eta} = \overline{\mu} \land 0 = x \right) \land \\
\end{align*} \]

\[ \hat{\nu} = \hat{\nu} \land x = x \land \overline{\nu} = \overline{\nu} \land \overline{\eta} = \overline{\mu} \land \overline{\nu} = \overline{\mu} : \overline{I} \]

Statement: For example, the disjuncts and are for each

Transitions Relation:

\[ 0 = \hat{\nu} = x \land 0^0 \mu = \overline{\nu} \land 0^0 = \overline{\mu} : \Theta \]

Initial condition:

\[ \begin{align*}
\{ \overline{\eta}, \overline{\mu}, 0^0 \nu \} : \overline{\nu} \\
\{ \overline{\eta}, 1^0, 0^0 \nu \} : \overline{\mu} \\
\text{natural} : \hat{\nu}, x \\
\end{align*} \]

State Variables:

The Corresponding FDS
Let $D$ be an FDS for which the above components have been identified. The state $s_0$ is defined to be a $D$-successor of state $s$ if the state $s = 0, 1, \ldots$, the state $s_{j+1}$ is a $D$-successor of the state $s_j$. We define a computation of $D$ to be an infinite sequence of states $s_0, s_1, s_2, \ldots$, satisfying the following requirements:

- **Initiality:** $s_0$ is initial, i.e., $s_0 \in \Theta$.
- **Justice:** For each $j \in J$, contains infinitely many $j$-positions.
- **Consecution:** For each $j \in J$, contains infinitely many $j$-positions.
- **Compassion:** For each $p, q \in C$, if $\emptyset$ contains infinitely many $p$-positions, it must also contain infinitely many $q$-positions.
- **Deductive Verification of Reactive Systems**, NYU, Fall, 2007

For each $j \in J$, $\emptyset \subseteq \Theta$. Let $\mathcal{A}$ be a computation of $D$.
Examples of Computations

The following computation of program ANY corresponds to the case that statement \( h_1 \) is executed before \( m_0 \).

The following computation corresponds to the case that statement \( h_1 \) is the first executed statement:

\[
\begin{align*}
\langle u : h, v : i, w : z, j : z, \varphi : \tau \rangle & \xrightarrow{0} \langle \varphi : \tau, h : i, v : z, w : z, j : z \rangle \\
& \xrightarrow{0} \langle h : i, v : z, w : z, j : z, v : i \rangle \\
& \xrightarrow{0} \langle h : i, v : z, w : z, j : z, v : i \rangle
\end{align*}
\]

In a similar way, we can construct for each \( u \) a computation that executes the body of statement \( n \) \( 0 \) times and then terminates in the final state.

The following computation of program ANY\(v\) corresponds to the case that program ANY gives rise to a set of computations of the FDS corresponding to a program \( P \).

\[
\begin{align*}
\langle h : i, v : z, w : z, j : z, \varphi : \tau \rangle & \xrightarrow{0} \langle \varphi : \tau, h : i, v : z, w : z, j : z \rangle \\
& \xrightarrow{0} \langle h : i, v : z, w : z, j : z, v : i \rangle \\
& \xrightarrow{0} \langle h : i, v : z, w : z, j : z, v : i \rangle
\end{align*}
\]
Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving. While we can delay termination of the program for an arbitrary long time, we cannot postpone it forever.

Thus, the sequence

\[
\begin{align*}
\cdots & \overset{l_3}{\longleftarrow} \langle 3 : f_3 \ ' 0 : x : 0 ; w : z _v \ ' 0 ; j : \nu \rangle \\
& \overset{l_2}{\longleftarrow} \langle 2 : f_2 \ ' 0 : x : 0 ; w : z _v \ ' 0 ; j : \nu \rangle \\
& \overset{l_1}{\longleftarrow} \langle 1 : f_1 \ ' 0 : x : 0 ; w : z _v \ ' 0 ; j : \nu \rangle \\
& \overset{l_0}{\longleftarrow} \langle 0 : f_0 \ ' 0 : x : 0 ; w : z _v \ ' 0 ; j : \nu \rangle
\end{align*}
\]

Thus, the sequence

cannot postpone it forever.

While we can delay termination of the program for an arbitrary long time, we...
Justice is not enough. You also need compassion.

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

The compassion set of this program consists of

\[ \{ y > 0 \wedge (at\text{-}m_3), (at\text{-}m_2), (at\text{-}c_3), y < 0 \wedge (at\text{-}c_2) \} \]  

\[ \langle I - y = 0; \forall y < 0 \rangle \]

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

\[ \begin{cases} 
\text{Release } y & : \text{m}_4 \\
\text{Critical } y & : \text{m}_3 \\
\text{Request } y & : \text{m}_2 \\
\text{Non-critical } y & : \text{m}_1 
\end{cases} \]

\[ \begin{cases} 
\text{Release } y & : \text{c}_4 \\
\text{Critical } y & : \text{c}_3 \\
\text{Request } y & : \text{c}_2 \\
\text{Non-critical } y & : \text{c}_1 
\end{cases} \]
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 which violates accessibility for process \( P_1 \). Duetotherequirementofcompassion

\[
\begin{align*}
\langle 0, m_0 \rangle & \xleftarrow{\_m_1} \langle 0, m_4 \rangle & \xleftarrow{\_m_4} \langle 0, m_3 \rangle \\
\langle 0, m_0 \rangle & \xleftarrow{\_m_1} \langle 0, m_1 \rangle & \xleftarrow{\_m_1} \langle 0, m_1 \rangle \\
\langle 1, m_2 \rangle & \xleftarrow{\_m_2} \langle 1, m_2 \rangle & \xleftarrow{\_m_2} \langle 1, m_2 \rangle \\
\langle 1, m_3 \rangle & \xleftarrow{\_m_3} \langle 1, m_3 \rangle & \xleftarrow{\_m_3} \langle 1, m_3 \rangle \\
\langle 1, m_3 \rangle & \xleftarrow{\_m_3} \langle 1, m_4 \rangle & \xleftarrow{\_m_4} \langle 1, m_3 \rangle \\
\langle 1, m_4 \rangle & \xleftarrow{\_m_4} \langle 1, m_1 \rangle & \xleftarrow{\_m_4} \langle 1, m_1 \rangle \\
\langle 1, m_2 \rangle & \xleftarrow{\_m_2} \langle 1, m_2 \rangle & \xleftarrow{\_m_2} \langle 1, m_2 \rangle \\
\langle 1, m_0 \rangle & \xleftarrow{\_m_0} \langle 1, m_0 \rangle & \xleftarrow{\_m_0} \langle 1, m_0 \rangle \\
\langle 1, m_4 \rangle & \xleftarrow{\_m_4} \langle 1, m_3 \rangle & \xleftarrow{\_m_3} \langle 1, m_3 \rangle \\
\langle 1, m_3 \rangle & \xleftarrow{\_m_3} \langle 1, m_4 \rangle & \xleftarrow{\_m_4} \langle 1, m_3 \rangle
\end{align*}
\]

Consider the state sequence:

- Critical section at \( \tau_2 \). Similar requirement for \( P_2 \).
- Accessibility: Whenever process \( P_1 \) is at \( \tau_2 \), it shall eventually reach it's
- Mutual Exclusion: No computation of the program can include a state in
- Should satisfy the following two requirements:

Program MUX-SEM

A. Pnueli
In the following, let \( b \) be a boolean expression, \( r \) be a natural variable, and \( S_1, S_2, \ldots, S_k \) be statements.

- If \( b \) is true, execution proceeds to \( S_1 \), otherwise to \( S_2 \).
- \( \text{if} \ b \ \text{then} \ S_1 \ \text{else} \ S_2 \) is a conditional statement.
- \( y := e \) is an assignment statement.
- \( \text{await} \ b \) is an await statement.
- \( \text{request} \ r \) is a request statement.
- \( \text{release} \ r \) is a release statement.
- \( \text{wait} \ q \) is an wait statement.
- Critical and Non-critical are schematic statements. They are used to denote sections in mutual-exclusion programs. They are used to denote sections in mutual-exclusion programs.
- For a variable \( y \) and an expression \( e \) of compatible type, \( y := e \) is an assignment statement.
- \( \text{Syntax} \): SPL
Lecture 1: Modeling Systems

A Pnueli

\[ S_1, S_2, \ldots, S_k \text{ is a concatenation statement. It executes } S_1, S_2, \ldots, S_k \text{ sequentially.} \]

\[ S_1 \text{ or } S_2 \text{ or } \ldots \text{ or } S_k \text{ is a selection statement. It non-deterministically chooses an enabled statement among } S_1, S_2, \ldots, S_k \text{ and proceeds to execute it.} \]

\[ \text{while } q \text{ holds, } S \text{ is a while statement. Statement } S \text{ is repeatedly executed as long as } q \text{ holds.} \]
Programs have the form 
\[
dec \; \text{statement};
\]
where each \( \text{P}_i \) is a process having the form 
\[
\text{declaration} \; \text{statement}; \ldots \; \text{declaration} \; \text{statement}
\]
A program has the form 
\[
\text{decl} \; \text{statement} \ldots \; \text{decl} \; \text{statement}
\]
A declaration consists of a sequence of declaration statements. Programs and processes may optionally be named.

A 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 imposes constraints on the initial values of the variables declared in this statement. Let \( \phi_1, \ldots, \phi_n \) be the assertions appearing in the declaration statements of a program. We refer to the conjunction \( \phi_1 \land \ldots \land \phi_n \) as the data-precondition of the program.

Let \( \phi \) be a formula. 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 imposes constraints on the initial values of the variables declared in this statement. Let \( \phi_1, \ldots, \phi_n \) be the assertions appearing in the declaration statements of a program. We refer to the conjunction \( \phi_1 \land \ldots \land \phi_n \) as the data-precondition of the program.
The Initial Condition
At this point, we take \( \Lambda = \emptyset \).

Observable Variables
For given locations \( l_j \) for \( j = 1, \ldots, m \), we write \( \text{at} l_j \) as an abbreviation for \( \forall \nu \in T \quad l_j = \nu \).

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 \( i \) which are declared at the head of the program and its processes, and range over their respectively declared data domains. The control variable \( i \) ranges over the location set of \( P \). For \( i = 1, \ldots, m \), the value of \( l_i \) in a state denotes the current location of control in the execution of process \( P_i \).

Observable Variables
At this point, we take \( O = V \).

The Initial Condition
Let \( \phi \) denote the data precondition of program \( P \).

State Variables
Let \( T \) denote the set of locations within process \( P \).

FDS corresponding to program \( P \), be a program. We proceed to construct the

Spl : Semantics

A. Pnueli
A. Pnueli

Let the initial condition for $D$ be:

\[ 1 = 01 \quad \land \quad m = 0^m \quad \land \quad '; \]

where, $0_i$ 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.

where, $\Theta$ is the initial condition of process $P_i$. Define the initial condition $\Theta$ for $\mathcal{D}$ as:

\[ \Theta : \bigvee \left( \Theta_0 \land \Theta_1 \land \cdots \land \Theta_n \right) \]
and contributes to the requirement $\mathcal{L}$ at $\gamma$. 

$$(\{\gamma, \Delta\} - \Delta) \pres \wedge \varepsilon = \delta \wedge \forall \gamma \at \gamma \delta \pres \wedge \forall \gamma \at \gamma \delta$$

For each type of statement, we indicate the disjunct contributed to the disjunct $d$.

**The assignment statement**

considered statement of the assignment statement $`j := y; e; k := y; e`$ contributes to the disjunct $\Lambda \subseteq \Omega$ stating that all the variables in the variable set $\Lambda$ are preserved by the $\forall \gamma \in \Omega \pres (\gamma = \delta)$ 

for each type of statement, we indicate the disjunct contributed to the disjunct $d$.

**Deductive Verication of Reactive Systems**, NYU, Fall, 2007
and contributes to the requirement \( J \) the requirement \( \mathsf{at} - \mathsf{at} \).

\[
(\{ \nu \} - \Lambda) \mathsf{press} \lor I + t = \nu \lor 0 < \nu \lor \mathsf{at} - \nu \lor \mathsf{at} - \nu
\]

\( j \) \( k \) contributes to the disjunct \( d \) to the requirement \( \mathsf{release} \) \( r \).

\[
(\{ \nu \} - \Lambda) \mathsf{press} \lor I - t = \nu \lor 0 < \nu \lor \mathsf{at} - \nu \lor \mathsf{at} - \nu
\]

\( j \) \( k \) contributes to the disjunct \( d \) to the requirement \( \mathsf{request} \) \( r \).

The release statement \( j \) \( k \) contributes to the requirement \( \mathsf{release} \) \( r \). which stays forever at \( j \) while \( q \) continuously holds.

\[
(\{ \nu \} - \Lambda) \mathsf{press} \lor q \lor \mathsf{at} - \nu \lor \mathsf{at} - \nu
\]

The request statement \( j \) \( k \) contributes to the requirement \( \mathsf{request} \) \( r \) contribution of \( q \) \( \mathsf{at} - \nu \lor \mathsf{at} - \nu \).
The statement `\(j: \text{Non-Critical}; k: \)` contributes to the disjunct at `\(j^\land 0\) \(k^\land \text{pres}(Vf_i)\)` and does not contribute any fairness requirement. This corresponds to the assumption that non-critical sections may fail to terminate.

The statement `\(j: \text{Critical}; k: \)` contributes to the disjunct `\(d\)` and critical sections must terminate.

In contrast to non-critical sections, critical sections must terminate. The conditional statement `\(\text{if } b \text{ then } S_1 \text{ else } S_2\)` contributes to the disjunct `\(d\)` if `\(q\)` and does not contribute any fairness requirement.

The statement `\(j: \text{Non-Critical}; k: \)` contributes to the disjunct `\(d\)` and does not contribute any fairness requirement.
can be attributed to one of their sub-statements. Any action performed by one of these statements contributes to one of their disjuncts of their own. The while statement \( j: \)

\[
\text{while } b \text{ do } \left[ S_1 \right] \;
\]

Note that the concatenation and selection statements do not contribute any disjunct of the requirement \( J \). Any action performed by one of these statements can be attributed to one of their sub-statements.

\[
\text{pres} : \; \text{Id}
\]

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

\[
\wp\; \text{pres} \vee \left( \{ ? \} - \text{pres} \right) \vee \left( \{ ? \} - \text{pres} \right) \vee \left( \{ ? \} - \text{pres} \right) \;
\]

The while statement \( j: \text{while } q \text{ do } \)

\[
\text{alt-} \;
\]

and contributes to the disjunct \( J \). The while statement \( j: \text{while } q \text{ do } \)

\[
\text{alt-} \;
\]

and contributes to the disjunct \( J \).
A model for a temporal formula is an infinite sequence of states $\sigma : \sigma_0, \sigma_1, \ldots$

- **Back-to, Weak-Since**
  \[(b S d) \land d \square = b \square d\]

- **Always in the past**
  \[d \lozenge \lozenge = d \square\]

- **Since**
  \[d S 1 = d \diamondsuit\]

- **Sometimes in the past**
  \[d S 1 = d \diamondsuit\]

- **Waiting-for, Unless**
  \[(b N d) \land d \square = b \bigwedge d\]

- **Henceforth**
  \[d \lozenge \lozenge = d \lozenge\]

- **Eventually**
  \[d \lozenge \lozenge = d \lozenge\]

- **Eventually**
  \[d \lozenge \lozenge = d \lozenge\]

- **Waiting-for, Unless**
  \[(b N d) \land d \square = b \bigwedge d\]

- **Henceforth**
  \[d \lozenge \lozenge = d \lozenge\]

- **Eventually**
  \[d \lozenge \lozenge = d \lozenge\]

- **Since**
  \[d S 1 = d \diamondsuit\]

- **Until**
  \[d S 1 = d \diamondsuit\]

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

- **Until**
  \[d S 1 = d \diamondsuit\]

- **Previous**
  \[d \bigodot\]

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

A requirement specification language is an underlying (first-order) assertion language. The predicate at $\xi$ abbreviates the formula $\forall \xi \exists \eta \psi$, where $\eta$ is a location within process $\xi$.
Given a model $$\mathcal{A}$$, we define the notion of a temporal formula holding at a position in $$\mathcal{A}$$, denoted by $$\mathcal{A} \models \varphi (j)$$.

\[ \mathcal{A} \models \varphi (j) \text{ if and only if } \mathcal{A} \models \varphi (j) \forall j \geq 0. \]

That is, we evaluate locally on state $$\mathcal{A}$$:

\[ \mathcal{A} \models \varphi (j) \text{ if and only if } \mathcal{A} \models \varphi (j) \forall j \geq 0. \]

Semantics of LTL

Deductive Verification of Reactive Systems, NYU, Fall, 2007
Formulas $p$ and $q$ are equivalent, denoted $p \equiv q$, if $p \iff q$ is valid. They are called congruent, denoted $p \simeq q$, if $(p \iff q)$ is valid. If $p \simeq q$ then $p$ can be replaced by $q$ in any context.

The entailment $\vdash b \iff d$ is an abbreviation for $b \simeq d \land (b \iff d)$.

Formulas and are equivalent, denoted $b \simeq d$ or $b \simeq d$. They are called congruent.
Following every \( b \), \( b \) precedes \( d \).

\[ \vdash b \rightarrow b \wedge (d \rightarrow d) \]

\[ \rightarrow \]

\[ \]

\[ \]

\[ \rightarrow \]

\[ \]

\[ \]

\[ \rightarrow \]

\[ \]

\[ \]

\[ \rightarrow \]e

\[ \]

\[ \]

\[ \rightarrow \]

\[ \]

\[ \]

\[ \rightarrow \]

\[ \]

\[ \]
A.Pnueli

Temporal Specification of Properties

Formula \( \varphi \) is \( \mathcal{D} \)-valid, denoted \( \mathcal{D} \models \varphi \), if all computations of \( \mathcal{D} \) satisfy \( \varphi \).

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 \( g_3 \) while \( P_2 \) is at \( m_3 \). Specifiable by the formula \((\text{at} \ g_3 \land \text{at} \ m_3) \neg (\varphi)\).

- ACCESSIBILITY for \( P_1 \)
- Whenever process \( P_1 \) is at \( g_2 \), it shall eventually reach its critical section at \( g_3 \). Specifiable by the formula \((\text{at} \ g_2) \rightarrow (\Diamond \ (\text{at} \ g_3))\).

- MUTUAL EXCLUSION
- No computation of the program can include a state in which process \( P_1 \) is at \( g_3 \) while \( P_2 \) is at \( m_3 \). Specifiable by the formula \((\text{at} \ g_3 \land \text{at} \ m_3) \neg (\varphi)\).

- Deductive Verification of Reactive Systems, NYU, Fall, 2007
Every propositional temporal formula can be translated into a first-order logic with monadic predicates over the naturals ordered by $\succ$.

For example, the first-order translation of $b \diamond \leftarrow d$ is:

\[
((\exists t) b) : 1_t \geq \exists t \in A_t \quad (\exists t) d : 0 \geq 1_t
\]
A formula of the form $p$ for some past formula $p$ is called a safety formula.

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

An equivalent characterization is the form $(p \quad \square \quad q)$. The equivalence is justified by $(p \quad \square \quad q)$.

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

$$(b \quad \square \quad \diamond \quad d \quad \diamond \quad \square) \quad \bigvee \quad \gamma$$

A formula of the form $d$ for some past formula $d$ is called a response formula.

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

Classification of Formulas/Properties
A. Pnueli

Hierarchy of the Temporal Properties

\[ (\exists b \diamond \bigwedge_{i=1}^{\exists} d \square (p_i \rightarrow q_i)) \]

\[ (\exists b \square \diamond \bigwedge_{i=1}^{\exists} d \diamond (p_i \rightarrow q_i)) \]

where \( d \), \( b \), \( q_i \) are past formulas.