Analysis of Reactive Systems, Algorithmic and Deductive Methods

Amir Pnueli

Thursdays, 5:00-6:50 PM

Textbooks:

Copies of presentations and lecture notes will be available at
http://www.cs.nyu.edu/courses/fall09/G22.3033-011/index.htm

Anaylsis of Reactive Systems, Algorithmic and Deductive Methods
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:

- Reactive systems and their specification by LTL, CTL, and CTL*.
- Deductive verification of infinite-state systems using theorem provers such as CVC, PVS, and STeP.
- Model checking of finite-state systems using BDD techniques over the TLV tool.
- Abstraction methods for combining deductive principles with algorithmic methods.

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

We distinguish two classes of programs:

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

Reactive Systems

Can be specified as a black box.

Specified in terms of Input/Output relations.

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

Can be specified by the requirement
\[ x = f(y) \]

A. Pnueli

Classification of Programs

Analysis of Reactive Systems, NYU, Fall 2009
Such programs must be specified and verified in terms of their behaviors.

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

Programs whose role is to maintain an ongoing interaction with their environments.
A computational model providing an abstract syntactic base for all reactive systems. We use Fair Discrete Structures (FDS).

Speciﬁcation Languages for specifying systems and their properties. We use temporal logics: \( \text{CTL}^*, \text{CTL}, \text{LTL} \) and \( \text{LTL} \).

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 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 framework for reactive systems veriﬁcation.
Fair Discrete Systems

A. Pnueli

Fair Discrete Systems

A fair discrete system consists of:

\( \{ \langle d_1, \ldots, d_k \rangle \} = \mathcal{D} \)
A Simple Programming Language: SPL

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

Consider the program

Example: Program ANY-Y

A Simple Programming Language: SPL

Lecture 1: Modeling Systems, NYU, Fall 2009
Compassion set: \( \emptyset \) •

Justice set: \( \mathcal{J} \) •

\[
\begin{align*}
\dot{h} &= \dot{h} \land x = \bar{x} \land \bar{v} = \bar{v} \land \\
& \quad \left( \bar{y} = \bar{v} \land 0 \neq x \right) \land \\
& \quad \left( \bar{y} = \bar{v} \land 0 = x \right) \land \\
& \quad 0 = \bar{v} : 0 d
\end{align*}
\]

\[
\begin{align*}
\dot{h} &= \dot{h} \land x = \bar{x} \land \bar{v} = \bar{v} \land \bar{v} = \bar{v} \land \\
& \quad \bar{v} = \bar{v} : 1 d
\end{align*}
\]

Statement. For example, the disjuncts and are appropriate disjunct for each with appropriate disjunct for each

\[
\begin{align*}
0 = \bar{v} = \bar{v} \land 0 = \bar{v} : 0 \land \\
\end{align*}
\]

Initial condition: \( \Theta \) •

State Variables: \( \forall \) •

The Corresponding EDS
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 $s$. Let $\mathcal{A}$ be an infinite sequence of states $s_0, s_1, s_2, \ldots$ satisfying the following requirements:

1. **Initiality:** $s_0$ is initial, i.e., $s_0 \in \Theta$.
2. **Consecution:** For each $j \geq 0, 1, \ldots$, the state $s_{j+1}$ is a $d$-successor of the state $s_j$.
3. **Justice:** For each $f \in \mathcal{F}$, $f$ contains infinitely many $f$-positions.
4. **Compassion:** For each $f \in \mathcal{F}$, $f$ contains infinitely many $f$-positions.

We define a computation of $\mathcal{A}$ to be an infinite sequence of states $\mathcal{A} = (s, s, \ldots)$.

For each $d$, $\mathcal{A}^d = (s, s, \ldots)$.

**Analysis of Reactive Systems**, NYU, Fall, 2009
Examples of Computations

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement $\text{statement}_0$ $n$ times and then terminates in the final state $\text{final state}$.

The following computation corresponds to the case that statement $\text{statement}_1$ is executed before $\text{statement}_0$.

The following computation of program $\text{program}_0$ corresponds to the case that $\text{statement}_0$ is the first executed statement.

The following computation corresponding to a program $\text{program}_d$ gives rise to a set of $\text{set}_d$.

Identification of the FDS $\text{FDS}_d$.

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

\[
\cdots \leftrightarrow \langle 0 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle
\]

\[
\leftrightarrow \langle 2 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle \leftrightarrow \langle 2 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle
\]

\[
\leftrightarrow \langle 1 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle \leftrightarrow \langle 1 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle
\]

\[
\leftrightarrow \langle 0 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle \leftrightarrow \langle 0 : f_i : 0 : x : 0 : m_0 : z : 0 : g : 0 : j : 1 : v \rangle
\]

Thus, the sequence cannot postpone it forever.

While we can delay termination of the program for an arbitrary long time, we

**A Non-Computation**
The following program \textsc{mux-sem}, implements mutual exclusion by semaphores.

\textbf{Justice is not Enough. You also Need Compassion}
Program MUX-SEM

Conclusions: Justice alone is not sufficient. For $J_2$, it is not a computation and accessibility is guaranteed. Which violates accessibility for process $P_1$. Due to the requirement of compassion.

Consider the state sequence:

`\[\langle 0, m_0, I \rangle \rightarrow \langle 0, m_3, I \rangle \rightarrow \langle 0, m_4, I \rangle \rightarrow \langle 0, m_3, I \rangle \rightarrow \ldots \]

which violates accessibility for process $P_1$. Duet to the requirement of compassion.

Conclusions: Analysis of Reactive Systems, NYU, Fall, 2009
sections in mutual-exclusion programs.

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

- Critical
  
  **Release** $r$ is a release statement. It increments $r$ by 1.
  
  **Request** $r$ is a request statement. It is enabled only when $r > 0$ and, when executed, it decrements $r$ by 1.

- Non-critical
  
  **Await** $q$ is an await statement. It waits for $q$ to become true, and then terminates.

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

- Schematic
  
  $S_1, \ldots, S_n$ be statements.

- In the following, let $q$ be a boolean expression, $r$ be a natural variable, and $S$ be statements.

**Syntax**

A.Pnueli
A. Pnueli

Lecture 1: Modeling Systems

Analysis of Reactive Systems, NYU, Fall, 2009

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 \( S_2 \).

- \( S_1;S_2;\ldots;S_k \) is a concatenation statement. It executes \( S_1 \); \( S_2 \); \( \ldots \); \( S_k \) sequentially.

- while \( b \) do \( S \) is a while statement. Statement \( S \) is repeatedly executed as long as \( b \) holds.

- \( S_1 \) or \( S_2 \) or \( \ldots \) or \( S_k \) is a selection statement. It non-deterministically chooses an enabled statement among \( S_1 \); \( S_2 \); \( \ldots \); \( S_k \) and proceeds to execute it.

- to \( S_1 \), otherwise to \( S_2 \).
Communication Statements

There are two communication statements:

- \( e \) is a send statement. It sends the value of expression \( e \) on channel \( x \).

- \( x \) is a receive statement. It reads a message from channel \( x \).

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

- \( \text{channel} \) — declares a synchronous channel which can transmit one message of type \( T \) at a time.

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

- \( \text{channel} \[ \cdot :: k \] \) — declares an asynchronous channel with \( k \)-bounded buffering capacity which can transmit messages (values) of type \( T \).
A program \( P \) has the form

\[
\text{declaration; statement}
\]

where each \( P_i \) is a process having the form

\[
\text{declaration; statement}
\]

A program and processes may optionally be named.

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

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

where \( \varphi \), \( \psi \), \ldots, \( \varphi \) are variables that share a common type and identify 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.

Each declaration statement lists several variables that share a common type and

\[
\varphi \land \cdots \land \varphi
\]

as the data-precondition of the program. We refer to the conjunction of the assertions appearing in the declaration statements of a program as the data-precondition of the program.

Let \( \varphi_1, \ldots, \varphi_n \) be the assertions declared in this statement.
Let \( P ::= \text{declaration} \); \( P_1 \ldots P_m \) be a program. We proceed to construct the FDS corresponding to program \( P \).

State Variables

For each declared channel \( a \) of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \). For each declared channel of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \).

For given locations \( \ell \), we write \( \text{at } a \) as an abbreviation for \( \nu \ell \). Let \( \ell \) denote the current location of control in the execution of process \( P \).

For each declared channel of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \).

State Variables

The state variables of system \( \mathcal{D} \) consist of the data variables \( \tau \) and the control variables \( c = 1, \ldots, m \), one for each process. The data variables \( \tau \) range over their declared data domains. The control variable \( c \) ranges over the location set of \( \mathcal{L} \), for \( c = 1, \ldots, m \).

The state variables of system \( \mathcal{D} \) consist of the data variables \( \tau \) and the control variables \( c = 1, \ldots, m \), one for each process. The data variables \( \tau \) range over their declared data domains. The control variable \( c \) ranges over the location set of \( \mathcal{L} \), for \( c = 1, \ldots, m \).

Let \( \ell \), \( \ell = 1, \ldots, \mathcal{L} \), denote the current location of control in the execution of process \( P \).

For each declared channel of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \).

State Variables

The state variables of system \( \mathcal{D} \) consist of the data variables \( \tau \) and the control variables \( c = 1, \ldots, m \), one for each process. The data variables \( \tau \) range over their declared data domains. The control variable \( c \) ranges over the location set of \( \mathcal{L} \), for \( c = 1, \ldots, m \).

For each declared channel of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \).

State Variables

The state variables of system \( \mathcal{D} \) consist of the data variables \( \tau \) and the control variables \( c = 1, \ldots, m \), one for each process. The data variables \( \tau \) range over their declared data domains. The control variable \( c \) ranges over the location set of \( \mathcal{L} \), for \( c = 1, \ldots, m \).

For each declared channel of type \( T \), we define variable \( a \) whose type is \( \mathbb{T} \).
Let $\varphi$ denote the data precondition of program $P$. We define the initial condition

$$\Theta = \varphi$$

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.

For each channel $\alpha$, $\varphi$ includes the conjunct $\alpha = \Lambda$, where $\Lambda$ denotes the empty list.
and contributes to the requirement $\mathcal{L}$ at $\mathcal{J}$.

\[(\{y \in U \} - \Lambda)\text{pres} \land \forall e = \bar{f} \land \forall y \in \Phi \text{ pres } \forall f \in \Omega \text{ pres}\]

The assignment statement $\text{at } \mathcal{J}$ contributes to the disjunct $\mathcal{J}$.

Considered statement.

Stating that all the variables in the variable set are preserved by the assignment.

\[\Lambda \subseteq \Omega \]

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

Belongs.

We denote by $P_i$ the process to which the considered statement contributes.

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 indicate the disjunct contributed to the transition relation, justice, and compassion, and the processes to which the statement contributes.
The \texttt{await} statement `\texttt{j: await b;}` contributes to the disjunct `(\{i \in I \} - \Lambda) \text{press} \lor I + i = i \lor \text{at}\texttt{\{f\} \textbf{alt}\texttt{\{f\}}} \lor \text{at}\texttt{\{f\}}`.

\textbf{The Release Statement:} `\texttt{j: release r;}` contributes to the disjunct `(\{i \in I \} - \Lambda) \text{press} \lor I - i = i \lor 0 < \text{r} \lor \text{at}\texttt{\{f\}} \lor \text{at}\texttt{\{f\}}`.

\textbf{The Request Statement:} `\texttt{j: request r;}` contributes to the disjunct `(\{i \in I \} - \Lambda) \text{press} \lor q \lor \text{at}\texttt{\{f\}} \lor \text{at}\texttt{\{f\}}`.

which stays forever \texttt{\{f\}} while \texttt{\{f\}} continuously holds.

\textbf{The Wait Statement:} `\texttt{j: await q;}` contributes to the disjunct `(\{i \in I \} - \Lambda) \text{press} \lor q \lor \text{at}\texttt{\{f\}} \lor \text{at}\texttt{\{f\}}`. 

\textbf{The Requirement:} \( J \), disallowing an execution which stays forever \texttt{\{f\}} while \texttt{\{f\}} continuously holds.
The statement \( j: \text{Non-Critical}; k: \text{contributesto the disjunct at } \nu \) and does not contribute any fairness requirement. This corresponds to the assumption that non-critical sections may fail to terminate.

\[
(\{?\} - \Lambda) \text{press} \lor \text{at-}^f \lor \text{at-}^c
\]

The statement \( j: \text{Critical}; k: \text{contributesto the disjunct at } \nu \). In contrast to non-critical and contributes to the requirement \( \forall \).

\[
(\{?\} - \Lambda) \text{press} \lor \text{at-}^f \lor \text{at-}^c
\]

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

The conditional statement

\[
\text{if } b \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.

The while statement

\[
\text{while } b \text{ do } S_1
\]

contributes to the disjunct $d$.

The conditional statement

\[
\text{if } q \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.

The conditional statement

\[
\text{if } q \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.

The while statement

\[
\text{while } b \text{ do } S_1
\]

contributes to the disjunct $d$.

The conditional statement

\[
\text{if } q \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.

The while statement

\[
\text{while } b \text{ do } S_1
\]

contributes to the disjunct $d$.

The conditional statement

\[
\text{if } q \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.

The while statement

\[
\text{while } b \text{ do } S_1
\]

contributes to the disjunct $d$.

The conditional statement

\[
\text{if } q \text{ then } S_1 \text{ else } S_2
\]

contributes to the disjunct $d$.
Asynchronous Communication

Let $a$ be an asynchronous channel with buffering capacity $k$, which is either a positive integer or the special symbol $\bot$ for the case of unbounded buffering. Let $\text{hd}$ and $\text{tl}$ be the head and tail list operations, respectively. It also contributes to $C$ the compass requirement:

$\text{at}_a \triangleright \text{hd} \land \text{at}_a \triangleright \text{tl}$. 

The asynchronous send statement $i \vdash e$ contributes to the disjunct

$(\{a, i\} - \Lambda)\text{pres} \lor (\varnothing)\mu = \varnothing \lor (\varnothing)\text{hd} = \varnothing \lor \forall i \text{ at}_a \triangleright \text{at}_a \triangleright \varnothing \lor \forall i \text{ at}_a \triangleright \varnothing$

The asynchronous receive statement $i \vdash x$ contributes to the disjunct:

$(\{a, i\} - \Lambda)\text{pres} \lor (\varnothing)\varnothing = \varnothing \lor (\varnothing)\text{hd} = \varnothing \lor \forall i \text{ at}_a \triangleright \text{at}_a \triangleright \varnothing \lor \forall i \text{ at}_a \triangleright \varnothing$

Let $a$ be an asynchronous channel with buffering capacity $k$, which is either a

Asynchronous Communication

A. Pnueli
A Pnueli
Lecture 1: Modeling Systems

Analysis of Reactive Systems, NYU, Fall 2009

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

Synchronous Communication
In addition to the above, the transition relation always contains the disjunct:

\[
(A \land \text{press}) : I_d
\]
A model for a temporal formula $\varphi$ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$.

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

- $\text{Strong Until}$
  - $p = \varphi U q$
  - $p = \varphi \rightarrow (p U q)$
  - $p = \varphi W q$
  - $p = \varphi S q$

- $\text{Strong Next}$
  - $p = \varphi X q$
  - $p = \varphi \rightarrow (p X q)$
  - $p = \varphi W X q$
  - $p = \varphi S X q$

- $\text{Strong Since}$
  - $p = \varphi F q$
  - $p = \varphi \rightarrow (p F q)$
  - $p = \varphi W F q$
  - $p = \varphi S F q$

A temporal formula is constructed out of state formulas (assertions) to which we apply the boolean operators $\land$, $\lor$, $\rightarrow$ and the basic temporal operators: $\varphi U \psi$, $\varphi X \psi$, $\varphi F \psi$, $\varphi S \psi$, $\varphi W \psi$, $\varphi W X \psi$, $\varphi S X \psi$, $\varphi W F \psi$, $\varphi S F \psi$.

Assume an underlying (first-order) assertion language $\mathcal{L}$. The predicate at $q_i$ abbreviates the formula $\forall q_i \in q_i$, where $q_i$ is a location within process $P_i$.

**Requirement Specification Language: Linear Temporal Logic**
A. Pnueli

Lecture 1
Modeling Systems

Semanticsof LTL

Given a model $\rho$, we define the notion of a temporal formula holding at a position $d$.

The semantics for the derived operators:

For all $d \models (\varphi, \rho)$

For some $d \models (\varphi, \rho)$

$\varnothing \models (\varphi', \rho)$

Semanticsof LTL
Lecture 1: Modeling Systems

A. Pnueli

If \( p \) holds over \( (s; 0) \), we say that \( p \) holds over and write \( \models s(0; \varnothing) \), if \( p \) is satisfiable if it holds over all models.

\( (b \leftrightarrow d) \square \) is an abbreviation for \( b \iff d \).

The entailment \( b \iff d \) is replaced by \( b \iff d \) in any context.

Formulas and are equivalent, denoted \( b \iff d \), if \( (b \iff d) \square \) is valid. They are called congruent, denoted \( b \iff d \), if \( (b \iff d) \square \) is valid. They are valid if valid.

\( \models s(0; \varnothing) \)
Following every $\phi$, $b$, $d$ precedes $\phi$.

Strongly precedes $b$.

Every is preceded by a causality.

Temporal Exercises:
- If $b$ holds at $s_0$, then $d$ holds at $s$ for some $s$.
- Such that $\phi$.

Reading Exercices
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:

\[
\neg (\text{at}_{t_3} \land \text{at}_{m_3})
\]

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:

\[
(\exists t' < t_3 \land \diamond\text{at}_{t'}) \land \square\text{at}_{t_3}
\]

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

\[ \phi \]

Formula \( \phi \) is \( \Delta \)-valid, denoted \( \Delta \vdash \phi \), if all computations of \( \Delta \) satisfy \( \phi \). Such a formula is \( \Delta \)-valid if and only if it is \( \Delta \)-valid for all runs.

Temporal Specification of Properties
This also shows that the past operators add no expressive power.

The logic $\mathcal{J}(\Box, \diamond)$. 

Claim 2. Every first-order formula can be translated into a temporal formula in $\mathcal{J}(\Box, \diamond)$.

Claim 1. Every first-order formula can be translated into a temporal formula in $\mathcal{J}(\Box, \diamond, \neg, S)$. 

The logic $\mathcal{J}(\Box, \diamond, \neg, S)$. 

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

\[ (((\exists t)b) : t \geq t \leftarrow (\forall t)d) : 0 \geq \exists t \diamond a \]

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

For example, the first-order translation of 0 is $b \diamond \leftarrow a$. 

Analysis of Reactive Systems, NYU, Fall, 2009
A formula of the form $\varphi$ for some past formula $\psi$ is called a safety formula.

A formula of the form $\psi$ for some past formula $\varphi$ is called a response formula.

An equivalent characterization is the form $\varphi \land \psi$.

Both formulas state that either there are infinitely many $\psi$’s, or there are no further $\varphi$’s, or there are no further $\psi$’s, beyond which there are no further $\varphi$’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.

\[
\varphi \land \psi
\]

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

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

By an equivalent characterization is the form $\varphi \land \psi$.

The equivalence is justified by the formula $\varphi \land \psi$.

A formula of the form $\varphi \land \psi$ for some past formula $\psi$ is called a response formula.

A formula of the form $\varphi \land \psi$ for some past formula $\psi$ is called a safety formula.

A formula of the form $\varphi \land \psi$ for some past formula $\psi$ is called a safety formula.

A formula of the form $\varphi \land \psi$ for some past formula $\psi$ is called a safety formula.