Course Outline

Course Contents

Classification of Programs

There are two classes of programs:

1. **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
     \[ y = x^2 + 3 + 2x \]

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

   - Can be specified by the requirement
     \[ x = y \]

   - The program which computes

   - Specified in terms of input/output relations.

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

Course Topics will cover:

- Expressions of the models and specification logics to real-time and hybrid systems.
- Methods for combining deductive principles with algorithmic techniques
- Proof of correctness of hybrid systems using theorem provers such as PVS and Step.
- Deductive verification of hybrid systems using the TITL tool.
- Systems and their specifications by TITL, model checking of finite-state systems.
- Main topics we will consider are:

  - Timed and hybrid systems.
  - Reactive systems.
  - Real-time systems.
  - Real-time, real-time, and Hybrid systems.

  - Repeat these for the model classes of reactive, real-time, and hybrid systems.

  - Study the Synthesis problem: Given a specification \( \phi \), construct a program \( p \) such that \( p \) satisfies \( \phi \), using theorem provers such as PVS and Step.

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

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

  - A.Pnueli

Lecture 1: Modeling Systems

I. Introduction to formal models of programs (designs) and their specifications.
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 green cactus (?) such programs must be specified and verified in terms of their behaviors.

A Hierarchy of Computational Models

- Reactive Systems
  - Expresses precedence relations between events

- Real-Time
  - Can measure temporal distance
  - Response: \( t_{\text{response}} \) \( \leq t_{\text{request}} \)

- 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:

Hard 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.

Soft real-time systems 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...
A.Pnueli

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.

\[ V = \{ x, y \} ;
\]
\[ O = \{ \text{next} \};
\]
\[ J = \{ \text{at} \};
\]
\[ C = \{ \text{next} \};
\]

- a set of \( \Theta \)-states.
- An initial condition, \( \Theta \) is a satisfiable assertion that characterizes the initial state.
- An initial condition, \( \Theta \) is a satisfiable assertion that characterizes the initial state.
- A set of \( \Theta \)-variables.
- A set of \( \Theta \)-variables.
- A set of \( \Theta \)-variables.

A language allowing composition of parallel processes communicating by shared variables as well as message passing.
The following program implements mutual exclusion by semaphores.

Justice is not Enough. You also Need Computation

A Non-Computation

Examples of Computations

\[ \left\{ \left\langle i : 1 \mid x : a, t y : b, t z : c \right\rangle \rightarrow \begin{cases} \text{critical} : y = y, & 1 + y = y, \\ \text{non-critical} : y > y, & 1 - y = y \end{cases} \right\} \]

The compassions set of this program consists of

\( \text{critical} \) and \( \text{non-critical} \)

The semaphore instructions \texttt{request} and \texttt{release} for respectively stand for

\[ \begin{bmatrix} \text{request} & : \text{wait} \\ \text{release} & : \text{wait} \end{bmatrix} \]

\[ \begin{bmatrix} \text{request} & : \text{wait} \\ \text{critical} & : \text{wait} \end{bmatrix} \]

\[ \begin{bmatrix} \text{request} & : \text{wait} \\ \text{non-critical} & : \text{wait} \end{bmatrix} \]

\[ \text{loop forever do} \]

Thus, the sequence cannot postpone it infinitely many -positions.

While we can delay termination of the program for an arbitrary long time, we must also contain infinitely many -positions. For each \( h, d \) we can consider the case that the body of statement \( n \), hence then conclusions in the final state of the body of statement \( n \), hence then conclusions in the final state executará.

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

\[ \begin{bmatrix} \text{request} & : \text{wait} \\ \text{non-critical} & : \text{wait} \end{bmatrix} \]

\[ \begin{bmatrix} \text{request} & : \text{wait} \\ \text{critical} & : \text{wait} \end{bmatrix} \]

\[ \text{loop forever do} \]

Computation: For each \( j, f \) contains infinitely many -positions.

Justice: For each \( j, f \) contains infinitely many -positions.

Satisfying the preceding requirements:

We define a computation of \( P \) to be an infinite sequence of states, if \( \delta \) is defined to be a -successor of state \( s \) and an initial state and an initial state.

\[ (\Delta, \Lambda)^0 = (s, s') \]

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

- **Asynchronous channel**: Declares a channel of type \( \text{channel}::k \) which can transmit messages of type \( k \) asynchronous channel.
- **Bounded buffering capacity channel**: Declares a channel \( \text{channel}[1..k] \) of type \( k \) which can transmit messages of type \( k \) with bounded buffering capacity.
- **Unbounded buffering capacity channel**: Declares a channel of type \( \text{channel}[1..\text{unbounded}] \) which can transmit messages of type \( \text{unbounded} \) without bounded buffering capacity.

There are 2 communication statements:

1. **Receive**: Reads a message from channel \( x \).
2. **Send**: Sends the value of expression \( e \) to channel \( x \).

There are 2 compound statements:

1. **While**: A statement that is repeatedly executed as long as \( b \) holds.
2. **If**: A selection statement that chooses an enabled statement among \( S_1, \ldots, S_n \).

### Communication Statements

- **Receive**: \( r \) is a receive statement. It reads a message from channel \( x \) if non-empty.
- **Send**: \( s \) is a send statement. It sends the value of expression \( e \) on channel \( x \).

### Compound Statements

- **While**: \( \text{while } b \) \{ \( S \) \} is a while statement. Statement \( S \) is repeatedly executed as long as \( b \) holds.
- **If**: \( \text{if } b \) \{ \( S_1 \) \} \text{ else } \{ \( S_2 \) \} is a selection statement. If \( b \) is true, execution proceeds to \( S_1 \); otherwise, to \( S_2 \).

### ProgramMUX-SEM

**Conclusion**: Justice above is not sufficient. For \( P \), if \( P \) is not a computation, and accessibility is guaranteed, which violates the requirement for process \( P \), due to the requirement of compassion:

- \( \text{MUX} \) is a computation, if \( P \) is at \( E \) and \( P \) is at \( E' \), whenever process \( P \) is at \( E \). It is not sufficient to include a statement in which process \( P \) is at \( E \), while \( P \) is at \( E' \).

**Mutual Exclusion**: When process \( P \) is at \( E \), no computation of the program can include a statement in which process \( P \) is at \( E \).

**Syntax**

\[ S\text{; } S_2 \ldots \text{; } S_n \]

**Semantics**

In the following, let \( \phi \) be a boolean expression, \( \alpha \) be a natural variable, and \( \beta \) be an expression.

- \( \alpha = e \) is an assignment statement.
- \( \text{while } \phi \Rightarrow S \) is an assignment statement.
- \( \text{if } \phi \Rightarrow S \text{ else } \{ S \} \) is an assignment statement.
- \( \text{while } \phi \Rightarrow \{ S \} \text{ else } \{ S \} \) is an assignment statement.
- \( \text{while } \phi \Rightarrow S \) is an assignment statement.
- \( \text{if } \phi \Rightarrow S \text{ else } \{ S \} \) is an assignment statement.
- \( \text{while } \phi \Rightarrow \{ S \} \text{ else } \{ S \} \) is an assignment statement.

**Critical and Non-critical sections in mutual-exclusion programs**

- Critical sections are statements that can be executed by only one process at a time.
- Non-critical sections are statements that can be executed by any number of processes.

**Critical section at \( P \)**: Similar requirement for \( P \).

When \( P \) is at a critical section, it is at a state where it is not accessible.

**Mutual Exclusion**:

- No computation of the program can include a statement in which process \( P \) is at a critical section.
The initial condition

\[ \text{let } \{ \text{initial condition} \} \text{ be the data precondition of program } P \text{.} \]

We define the initial condition as

\[ \text{let } \{ \text{initial condition} \} \text{ be the data precondition of program } P \text{.} \]

Each declaration statement consists of a sequence of declaration statements of the form

\[ \text{declaration: statement} \]

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

\[ \text{declaration: statement} \]

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

\[ \text{declaration: statement} \]

Each declaration statement consists of a sequence of declaration statements of the form

\[ \text{declaration: statement} \]
Asynchronous Communication

The statement $\text{send}(x)$ is always true. The synchronous send statement $\text{send}(x)$ contributes to $d$.

Let $\mathcal{V}$ be an asynchronous channel with buffering capacity $\ell$ which is either a positive integer or the special symbol $\infty$ for the case of unbounded buffering.

Note that the condition $|\ell| > 0$ guarantees that the requirement $d$ is satisfied by appending the expression $\Delta \mathcal{V}$ to $\ell$. This statement also contributes to the disjunct $\text{send}(x)$.

Compound Statements

The statement $\text{request}(x)$ is always true. The asynchronous request statement $\text{request}(x)$ contributes to $d$.

Assumption that non-critical sections may fail to terminate.

$\text{release}(x)$ and $\text{request}(x)$ are the head and tail list operations, respectively.

$\text{release}(x)$ contributes to $d$.
A model for a temporal formula \( \phi \) is an infinite sequence of states \( \langle s_0, s_1, \ldots \rangle \).

Given a model \( \phi \), we define the notion of a temporal formula holding at a position in a \( d \) denoted by \( d \models \phi \).

\[ d \models \phi \]

This implies the following semantics for the derived operators:

\[ d \models (\phi \lor \psi) \iff d \models \phi \lor d \models \psi \]

\[ d \models (\phi \land \psi) \iff d \models \phi \land d \models \psi \]

\[ d \not\models (\phi \land \psi) \iff d \not\models \phi \lor d \not\models \psi \]

\[ d \models \neg \phi \iff d \not\models \phi \]

\[ d \models\exists ! \psi \]

\[ d \models\forall \psi \]

\[ d \models A \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' \leq d \]

\[ d \models E \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models U \psi \iff \exists \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models O \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

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

\[ d \models F \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models G \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models L \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models W \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models H \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models T \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models R \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models X \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models U \psi \iff \exists \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models O \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models F \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models G \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models L \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models W \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models H \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models T \psi \iff \forall d' \models \psi \text{ such that } d' \models \psi \text{ for all } d' < d \]

\[ d \models R \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]

\[ d \models X \psi \iff \exists d' \models \psi \text{ such that } d' \models \psi \text{ and for all } d' < d \]
Following are some temporal formulas $\varphi$ and what do they say about a sequence $\sigma = \alpha_0, \alpha_1, \ldots$ such that $\sigma = \varphi$: 

- $p \implies q$: Every $p$ is preceded by a $q$. Can also be written as $\square p \implies q$. 
- $p \equiv q$: $p \implies q$ and $q \implies p$. The sequence $\sigma$ contains infinitely many $q$’s. 
- $p \Leftrightarrow q$: $p$ and $q$ are equivalent. 
- $p \implies q$: Every $p$ is followed by a $q$. 
- $p \equiv q$: $p$ and $q$ are equivalent. 
- The sequence $\sigma$ contains infinitely many $q$’s.

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

Note that $q$ is not guaranteed, but $q$ — precedence, $q$ cannot happen without a preceding $q$.

Every $q$ is preceded by a $p$ — causally.

Every $p$ is followed by a $q$.

Claim 1. Every 1st-order formula can be translated into a temporal formula in the logic $\text{L}$. 

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

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

Every (propositional) temporal formula $\varphi$ can be translated into a first-order logic with monadic predicates over the natural numbers ordered by $<$(1st-order theory of linear order). For example, the 1st-order translation of $p \implies q$ is $\exists x, x > 0, (x, x + 1)$.

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

W. Kamp [Kamp88] has shown that if we only allow addition and multiplication in our temporal formulas. But then proceeded to show that:

\begin{itemize}
  \item Mutual Exclusion — No computation of the program can include a state in which process $P_1$ is at $\ell_1$ while $P_2$ is at $\ell_2$. Specifiable by the formula $\neg \exists \alpha : \ell_1 \wedge \alpha = \ell_2$. 
  \item Accessibility for $P_1$ at $\ell_1$ — Whenever process $P_1$ is at $\ell_1$, it shall eventually reach its critical section at $\ell_1$. Specifiable by the formula $\exists \alpha : \ell_1 \wedge \alpha = \ell_1$.
\end{itemize}
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.

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

An equivalent characterization is the form $\forall i \exists j (b \odot \Diamond_i \land d \odot \Box_j)$. The equivalence is justified by the following formulas:

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

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

Classification of Formulas/Properties