In the next several lectures, we will present methods for model checking these properties.

Considered two classes of properties:

- **The assertion property** \( \text{assertion} \): claiming that all reachable states satisfy the assertion.
- **The invariance property** \( \text{inv} \): claiming that all \(-\text{reachable}\) states are all \(-\text{reachable}\).

Up to now, we have assumed a finite-state FDS presented by its components. Recall that a state

\[ b \] must be followed by a \(-state.\]

\[ b \psi d \Rightarrow \neg d \]

\[ \text{The response property} \]

\[ d \psi b \Rightarrow d \]

\[ \text{Assume a finite-state FDS presented by its components.} \]

Assume a finite-state FDS presented by its components.

- **The assertion property** \( \text{assertion} \): claiming that all reachable states satisfy the assertion.
- **The invariance property** \( \text{inv} \): claiming that all reachable states are all reachable.

Up to now, we have assumed a finite-state FDS presented by its components. Recall that a state

\[ b \] must be followed by a \(-state.\]

\[ b \psi d \Rightarrow d \]

\[ \text{The response property} \]

\[ d \psi b \Rightarrow d \]

\[ \text{Assume a finite-state FDS presented by its components.} \]

Assume a finite-state FDS presented by its components.

- **The assertion property** \( \text{assertion} \): claiming that all reachable states satisfy the assertion.
- **The invariance property** \( \text{inv} \): claiming that all reachable states are all reachable.
Model Checking

This is a process by which we algorithmically check that a given finite state \( D \) satisfies its specification presented as a set of properties. There are two approaches to this process:

- **Enumerative (explicit state)** approach, by which we construct a graph containing all the reachable states of the system, and then apply graph theoretic algorithms to its analysis.

- **Symbolic** approach, by which we consistently work with assertions which characterize sets of states.

We will start with the enumerative approach and then proceed with the symbolic approach. For comparison, the enumerative approach can handle systems with up to \( 10^6 \) states, while the symbolic approach can, on a good day, handle systems with up to \( 10^{1000} \) states.
A state-transition graph \((S; E)\) is a directed graph whose nodes are states of some system and whose edges connect state to state if there is a successor of \(s\).

\[ (d^0, s^0) \]

Initially place in \(S\) all states that are in \(S^0\).

Repeat the following step until no new states or new edges can be added.

\[ \text{Step: for some } s \in S, \text{ let } s^1, \ldots, s^k \text{ be the } d\text{-successors of } s. \text{ Add to } (d^0, s^0) \text{ all states among } \{s^1, \ldots, s^k\} \text{ which are not already there and add to } E \text{ edges connecting } s \text{ to } s^1, \ldots, s^k. \]

Return \((S, E)\) edges connecting to \(S \) to \(S^1, \ldots, S^k\) all states among \(S\) which are not already there and add to \(E\) edges connecting to \(S\) to \(S^1, \ldots, S^k\).

Algorithm \textsc{construct-graph}

\[ (d^0, s^0) \]

The state-transition graph contains all the states reachable from the set \(S^0\) by \(d\)-transitions.

\[ (d^0, s^0) \]

The following algorithm constructs the state-transition graph of \(S\) of some system and whose edges connect state to state if there is a successor of \(s\).

\[ (d^0, s^0) \]

\[ (d^0, s^0) \]
Below, we present a simpler version of program MUX-SEM.

Example: a Simpler MUX-SEM
Following is the state-transition graph for \textsc{mux-sem}. This graph contains all the states accessible by \textsc{mux-sem}. Here and elsewhere, we denote by the set of states satisfying \( \{ (N_1^1, N_2^1) \} \) is the set of initial states of \textsc{mux-sem}. Thus, \( d \parallel d \parallel \Theta \) for \textsc{mux-sem}.
Using this algorithm, we can ascertain that program \textsc{mux-sem} satisfies the

\textbf{Algorithm} MC-\textsc{Inv}(\textsc{d},\alpha) — verify that system \textsc{d} satisfies the invariance property \alpha

\begin{itemize}
  \item \textbf{Property Inv(\textsc{d},\alpha)}.
  \item \textbf{Algorithm MC-Inv(\textsc{d},\alpha)}.
  \item \textbf{Model Checking Invariance Properties}
  \item \textbf{Approaches to Model Checking}
  \item \textbf{Invariance Properties}
\end{itemize}
We iterate as follows:

**Iteration 1:**

- From the state $0, 2^1_N$, we transition to $1, 2^1_N$.
- From $1, 2^1_N$, we transition to $1, L^1_N$.
- From $1, L^1_N$, we transition to $1, 2^1_N$.
- From $1, 2^1_N$, we transition to $0, 2^1_N$.

**Iteration 2:**

- From the state $0, 2^1_N$, we transition to $1, 2^1_N$.
- From $1, 2^1_N$, we transition to $1, L^1_N$.
- From $1, L^1_N$, we transition to $1, 2^1_N$.
- From $1, 2^1_N$, we transition to $0, 2^1_N$.

**Illustrate Forward Exploration on MUX-SEM**
This last iteration has an empty intersection with $C_1 \cup C_2$. We conclude:

$$\text{Inv: } (C_1 \cup C_2).$$
Next, we consider the symbolic approach. Note that every assertion over a finite-domain FDS can be represented as a boolean formula over boolean variables including the initial condition $\Theta$ and the bi-assertion representing the transition relation.

A key development for symbolic model checking was the development of binary decision diagrams (BDD) as an efficient representation of boolean assertions.
In general, it requires an exponential number of nodes.

We start with a binary decision diagram. For example, following is a decision diagram (tree) for the formula \((\exists x = x) \lor (\forall x = \neg x)\):
Optimize

Identify identical subgraphs.

Yielding:

- Remove redundant tests.
- Identify identical subgraphs.

A Pnueli
For simplicity, we will refer to ROBDD simply as BDD.

- No redundant tests – \( \text{low}(n) \neq \text{high}(n) \) for all nodes \( n \) in the graph.
- Uniqueness – no two distinct nodes are the roots of isomorphic subgraphs.

A BDD is ordered (OBDD) if the variables respect a given linear order on all paths through the graph. An OBDD is reduced (ROBDD) if it satisfies:

- Uniqueness – no two distinct nodes are the roots of isomorphic subgraphs.
- No redundant tests – \( \text{low}(u) \neq \text{high}(u) \) for all nodes \( u \) in the graph.

A set of variables \( n \) of out-degree 2. The two outgoing edges are given by the functions \( \text{low}(n) \) and \( \text{high}(n) \). A variable \( \text{var}(n) \) is associated with each node.

A ROBDD is a rooted, directed acyclic graph with two or more nodes at out-degree zero (leaves) labeled 0 or 1, and one or two nodes at out-degree one.

A binary decision diagram BDD is a connected, rooted, directed acyclic graph with

Definitions
Claim 1. For every function $f : \text{Bool}^n \rightarrow \text{Bool}$ and variable ordering $x_1 > x_2 > \ldots > x_n$, there exists exactly one BDD representing this function.
Sensitivity to Variable Ordering

The complexity of BDD representation is very sensitive to the variable ordering. For example, the BDD representation of \((x_1 = y_1) \land (x_2 = y_2)\) under the variable ordering \(x_1 < x_2 < y_1 < y_2\) is:
Implementation of BDD Packages

Types and Variables:

\[
\begin{align*}
\text{node} & : \text{nat} \\
\text{var} & : \text{nat} \\
\text{low} & : \text{nat} \\
\text{high} & : \text{nat} \\
\text{var-num} & : \text{nat} \\
\text{naturals} & : \text{set} \\
\text{record} & : \text{nat} \\
\text{init} & : \text{nat} \\
\text{new} & : \text{nat} \\
\text{H} & : \text{nat} \\
\text{T} & : \text{nat} \\
\text{f?g} & : \text{nat} \\
\end{align*}
\]

Operations:

\[
\begin{align*}
\text{init} & \left( \langle n \rangle \right) = (n)T \\
\text{new} & = n \\
\text{init} & \left( \langle n \rangle \right) = (n)H \\
\end{align*}
\]

Hardware Verification, NYU, Fall, 2004
A. Pnueli

Internal Representation

Lecture 4: Approaches to Model Checking

Hardware Verification, NYU, Fall 2004
A.Pnueli

Making or Retrieving a Node

Function $\text{M}(i, j, n, \text{num}, \text{var}, \text{id}, \text{node})$

--- Making or Retrieving a Node with Attributes $(i, j, n)$

\begin{align*}
  \text{return } n & : (j, i, n)H \\
  n & := (j, i, n)H \\
  (j, i, n) & := n \\
  \text{if } H(i, j, n) & \neq \top \text{ then return } j \\
  j & := i \\
  \text{if } \text{then return } \text{return } j \\
\end{align*}

Hardware Verification, NYU, Fall, 2004
Let \( \text{op} : \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \) be a binary boolean operation. The following function uses the auxiliary dynamic array \( G \):

\[
\text{Apply}(\text{op}; u_1; u_2) \rightarrow \text{node} : \begin{cases} 
\text{node} : \mathbb{2} & \text{if } \text{op}(u_1; u_2) \in \{1, 0\} \\
\text{node} \times \text{node} & \text{if } \text{op}(u_1; u_2) \notin \mathbb{2}
\end{cases}
\]

\[
\text{for } i \in [\mathbb{2}; \mathbb{1}] \text{ do } 
\text{node} : \mathbb{2} \text{ if } \text{var}(u_1) = \text{var}(u_2) \\
\text{node} : \mathbb{2} \times \text{node} \text{ else if } \text{var}(u_1) < \text{var}(u_2) \\
\text{node} : \mathbb{1} \times \text{node} \text{ else if } \text{var}(u_1) > \text{var}(u_2)
\]

\[
\text{function App}(u_1; u_2) : \begin{cases} 
\text{node} : \mathbb{2} & \text{if } \text{var}(u_1) = \text{var}(u_2) \\
\text{node} : \mathbb{2} \times \text{node} & \text{if } \text{var}(u_1) < \text{var}(u_2) \\
\text{node} : \mathbb{1} \times \text{node} & \text{else if } \text{var}(u_1) > \text{var}(u_2)
\end{cases}
\]

\[
\text{return } \text{App}(u_1; u_2)
\]

\[
\text{end App}
\]

\[
\text{return } n
\]

\[
n =: [\mathbb{2}; \mathbb{1}]
\]

\[
([\mathbb{2}; \mathbb{1}]) \text{ App} =: n
\]

\[
([\mathbb{1}; \mathbb{2}]) \text{ App} =: n
\]

\[
\text{else if } \text{var}(u_1) = \text{var}(u_2)
\]

\[
\text{node} : \mathbb{2} 
\]

\[
\text{node} : \mathbb{2} \times \text{node} 
\]

\[
\text{node} : \mathbb{1} \times \text{node} 
\]

\[
\text{else if } \text{var}(u_1) < \text{var}(u_2)
\]

\[
\text{node} = \text{App}(u_1; \text{low}(u_2)) \\
\text{node} = \text{App}(u_1; \text{high}(u_2))
\]

\[
\text{else if } \text{var}(u_1) > \text{var}(u_2)
\]

\[
\text{node} = \text{App}(\text{low}(u_1); u_2) \\
\text{node} = \text{App}(\text{high}(u_1); u_2)
\]

\[
T =: G
\]
Restriction is the same as substitution. We denote by \( [q \leftarrow x] \mathcal{T} \) the result of substituting \( q \) for \( x \) in assertion \( \mathcal{T} \).

Restriction (Substitution)
Quantification

Existential quantification can be computed, using the equivalence:

$$[I \leftarrow x] t \land [0 \leftarrow x] t \sim t : \exists x$$

Universal quantification can be computed dually:

$$[I \leftarrow x] t \lor [0 \leftarrow x] t \sim t : \forall x$$