Analysis of Reactive Hardware Systems

Amir Pnueli


Available at http://www.cs.nyu.edu/courses/spring04/G22.3033-017/index.htm

Copies of presentations and lecture notes will be

Wednesdays, 7:00-8:50 PM

Hardware Verification, NYU, Fall, 2004
Course grades will be determined based on assignments and a term project.

- Verification of arithmetic units.
- Verification of microprocessors: Out-of-order executions and speculation.
- SAT Methods, bounded model checking, and extensions.
- Model reduction by symmetry and abstraction.
- State systems using BDD techniques.
- Systems and their specification by CTL, CTL*, and LTL, model checking of finite-state systems.

Course Outline

1. Modeling Systems

The course will focus on hardware verification. No prior deep understanding of hardware design principles is required beyond basic principles. The reason we concentrate on verification of hardware designs is that it is a large, interesting, and industrially motivated class of finite-state systems. The course will be dedicated to methods for the verification of large finite-state systems. The main topics we will consider are:

- Systems and their specification by CTL, CTL*, and LTL, model checking of finite-state systems.
There are two classes of programs:

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

The program which computes $y = 1 + 3 + \cdots + (2x - 1) + 1 = f_i$ can be specified by the requirement $y = x^2$.

**Example:** Specified in terms of Input/Output relations. Can be modeled as a black box.

The programs which can be modeled as a black box run in order to produce a final result on termination.
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.

Reactive Programs

Can be viewed as a green cactus (?)
A computational model providing an abstract syntactic base for all reactive systems.

A Framework for Reactive Systems Verification

Verification Techniques for validating that an implementation satisfies a specification. Practiced approaches:

- SMV input language for hardware systems description.
- An implementation language for describing proposed implementations (both software and hardware). Use SP, a simple programming language and the \texttt{SPL}.
- Temporal logics: \texttt{CTL}, \texttt{CTL*}, \texttt{SCTL}, and \texttt{LTL}.

Temporal logic for specifying systems and their properties. We use

A specificational language for specifying reactive systems and their properties. We use

systems. We use fair discrete structures (FDS).

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

A computational model providing an abstract syntactic base for all reactive systems.

A Specification Language for specifying systems and their properties. We use temporal logics: \texttt{CTL}, \texttt{CTL*}, \texttt{SCTL}, and \texttt{LTL}.

A Framework for Reactive Systems Verification
Fair Discrete Systems

A fair discrete system (FDS) consists of:

\[ \langle \mathcal{C}, \mathcal{L}, d, \Theta, \Lambda \rangle = \mathcal{D} \]

- A set of \( \mathcal{C} \)-states
- A set of \( \mathcal{L} \)-states
- A transition relation
- A satisfied condition
- An initial condition
- The set of all \( \Lambda \)-states

\( \mathcal{C} \) consists of:

\[ \{ \langle \mathcal{U}, \mathcal{V} \rangle, \cdots, \langle \mathcal{U}, \mathcal{V} \rangle \} = \mathcal{C} \]

\( \mathcal{L} \) consists of:

\[ \{ \langle \mathcal{I}, \mathcal{J} \rangle, \cdots, \langle \mathcal{I}, \mathcal{J} \rangle \} = \mathcal{L} \]

\( \mathcal{D} \) consists of:

\[ \{ \langle \mathcal{U}, \mathcal{V}, d, \mathcal{I} \rangle, \cdots, \langle \mathcal{U}, \mathcal{V}, d, \mathcal{I} \rangle \} = \mathcal{D} \]

A fair discrete system has infinitely many \( \mathcal{I} \)-states for each \( \mathcal{J} \)-state. Ensure that a computation has infinitely many \( \mathcal{J}_i \)-states for each \( \mathcal{J} \)-state. For example, \( 1 + x = 1 \)

\( x \) corresponds to the assignment to the state variables. For example, \( (\mathcal{L} \mathcal{L} \mathcal{L}) \) referring to both unprimed (current) and primed (next) versions of the state variables. An assertion referring to both unprimed and primed states.

- A satisfied condition
- An initial condition
- The set of all \( \Lambda \)-states

A fair discrete system (FDS) consists of:

\[ \langle \mathcal{C}, \mathcal{L}, d, \Theta, \Lambda \rangle = \mathcal{D} \]

A fair discrete system (FDS) consists of:

\[ \langle \mathcal{C}, \mathcal{L}, d, \Theta, \Lambda \rangle = \mathcal{D} \]
A Simple Programming Language: SPL

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

Example: Program ANY

Consider the program

\[
\begin{align*}
\begin{array}{c}
\text{natural initially} \\
\text{initially}
\end{array}
\begin{array}{c}
x = y = 0 \\
0 = \forall_i = x \forall_j
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{while} \quad \begin{array}{c}
x = 0 \\
x = y + 1
\end{array} \\
\text{do}
\end{align*}
\]

\[0 = \forall_i = x, \forall_j = y\]

Hardware Verification, NYU, Fall, 2004
Compassion set: \( C : \emptyset \)

Justice set: \( \mathcal{L} \)

\[
\begin{aligned}
\hat{r} = \hat{r} \lor x = x \lor \overline{z} u &= \overline{z} u \lor \\
&\left( \overline{z} y = \overline{1} u \lor 0 \neq x \right) \lor \\
&\overline{1} y = \overline{1} u \lor 0 = x)
\end{aligned}
\]

Initial condition:

\[
\begin{aligned}
0 = \hat{r} = x \lor \overline{0} w &= \overline{z} u \lor \overline{0} y = \overline{1} u : \Theta
\end{aligned}
\]

State Variables

Transition Relation: For example, the disjuncts and \( 0 d p \lor 1 d p \lor 0 d p \lor 1 d p \) are appropriate disjunct for each.

Initial condition:

\[
\begin{aligned}
\left( \begin{array}{c}
\{ \overline{1} w, \overline{0} w \} : \overline{z} u \\
\{ \overline{z} y, \overline{1} y, \overline{0} y \} : \overline{1} u \\
\text{natural} : \hat{r}, x
\end{array} \right)
\end{aligned}
\]
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 the state $s$. We define a computation of $D$ to be an infinite sequence of states $s_0; s_1; s_2; \ldots$. Let $\Sigma$ be an FDS for which the above components have been identified. The computations $\langle s, \Sigma \rangle$ must also contain initially many $\Sigma$-positions.

For each $s \in C$, if $s$ contains initially many $D$-positions, then $\langle s, D \rangle$ is a $D$-successor of the state $s$.

Compassion: For each $\theta \in C$, if $\theta$ contains initially many $D$-positions, then $s_0$ is initial, i.e., $s_0 \models \theta$.

Justice: For each $s \in F$, if $s$ contains infinitely many $D$-positions, then $s$ contains infinitely many $\Sigma$-positions.

Consecution: For each $j = 0, 1, \ldots$, the state $s_{j+1}$ is a $D$-successor of the state $s_j$.

Initiality: $s_0$ is initial, i.e., $s_0 \models \theta$.

Satisfying the following requirements:

$\phi : s_0; s_1; s_2; \ldots$.
Examples of Computations

Identification of the FDS corresponding to a program gives rise to a set of computations of the form:

\[(d, Q) \in \text{comp} = (P, d) \in \text{comp}\]

Lecture 1: Modeling Systems

Hardware Veri®cation, NYU, Fall, 2004
While we can delay termination of the program for an arbitrary long time, we cannot postpone it forever.

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

Thus, the sequence

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

\[
\begin{align*}
\{ y < 0 \land z < m \} : \{ C \}\text{.}
\end{align*}
\]

The compassion set of this program consists of

\[
\begin{align*}
\text{\texttt{wait}} & : y < 0 \land y > 0 \\
\text{\texttt{request}} \ y & : y =: y + 1 && \text{and} \quad \langle I - y =: y < 0 \land y > 0 \rangle
\end{align*}
\]

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

\[
\begin{align*}
- \quad P_2 \quad - \\
- \quad P_1 \quad -
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_4 \\
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_4
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_3 \\
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_3
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_2 \\
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_2
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_1 \\
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_1
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_0 \\
\begin{bmatrix}
\text{Release} \ y \\
\text{Critical} \ y \\
\text{Request} \ y \\
\text{Non-critical}
\end{bmatrix}
: m_0
\end{align*}
\]

The following program

\[
\begin{align*}
\text{\texttt{P}_1} \quad - \\
\text{\texttt{P}_2} \quad - \\
\text{\texttt{P}_1} \quad -
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{I}} & = y \quad \text{initially \ natural}
\end{align*}
\]

Justice is not enough. You also need compassion.
Program MUX-SEM

should satisfy the following two requirements:

- **Mutual Exclusion** – No computation of the program can include a state in which process $P_1$ is at $\ell_3$ while $P_2$ is at $m_3$.

- **Accessibility** – Whenever process $P_1$ is at $\ell_2$, it shall eventually reach its critical section at $\ell_3$. Similar requirement for $P_2$.

Consider the state sequence:

$$\sigma:\begin{align*}
\langle \ell_0, m_0, 1 \rangle \\
\langle \ell_2, m_3, 0 \rangle \\
\langle \ell_2, m_0, 1 \rangle \\
\langle \ell_2, m_3, 0 \rangle \\
\langle \ell_2, m_2, 1 \rangle \\
\langle \ell_2, m_2, 1 \rangle \\
\end{align*}$$

which violates accessibility for process $P_1$. Due to the requirement of compassion for $\ell_2$, it is not a computation, and accessibility is guaranteed.

Conclusion: Justice alone is not sufficient!!!
sections in mutual-exclusion programs.

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

  - Critical

  - Non-critical

  - Critical

  - Non-critical

- SPL: Syntax

- Syntax
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 $S_1; \ldots; S_k$ sequentially.
- while $q$ holds. long as $q$ holds, the while statement $S$ is repeatedly executed as while $q$ do $S$ and proceeds to execute it.
- if $q_1$, $q_2$, $\ldots$, $q_n$ and proceeds to execute it.
- if $q_1$, $q_2$, $\ldots$, $q_n$ and proceeds to execute it.

Selection Statement is a selection statement. It non-deterministically chooses an enabled statement among $S_1; \ldots; S_k$ and proceeds to execute it.

Selection Statement is a selection statement. It non-deterministically chooses an enabled statement among $S_1; \ldots; S_k$ and proceeds to execute it.
There are two communication statements:

- \( (e \text{ is a send statement. It sends the value of expression } e \text{ onto channel } c) \)
- \( (x \text{ is a receive statement. It reads a message from channel } c) \)

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

- \( \text{channel} \) of \( \cdot \) — declares a synchronous channel which can transmit one message of type \( \cdot \) at a time.
- \( \text{channel} \) of \( [1..k] \) of \( \cdot \) — declares an asynchronous channel with bounded buffering capacity which can transmit messages (values) of type \( k \) which can transmit messages (values) of type \( \cdot \) with \( k \) -bounded buffering capacity which can transmit messages (values) of type \( \cdot \).
- \( \text{channel} \) of \( \cdot \) — declares an asynchronous channel with unbounded buffering capacity which can transmit messages (values) of type \( \cdot \).

Hardware Verification, NYU, Fall, 2004
A program $P$ has the form

$$P = \langle \text{declaration}; \text{statement} \rangle$$

where each $P_i$ is a process having the form

$$P_i = \langle \text{declaration}; \text{statement} \rangle$$

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

$$\phi \quad \phi \quad \cdots \quad \phi$$

where $\phi$ are where variables, and $\phi$ are where variables.

A declaration starts with each of the variables of the program's initial values of the variables declared in this statement. The optional assertion $\phi$ imposes constraints on the variables' initial values. The optional assertion $\phi$ imposes constraints on the initial values of the variables declared in this statement.

Let $\phi, \phi, \cdots, \phi$ be the assertions appearing in the declaration statements of a program. We refer to the conjunction $\phi \lor \cdots \lor \phi$ as the data-precondition of the program.

Programs and processes may optionally be named.

Hardware Verification, NYU, Fall, 2004

Lecture 1: Modeling Systems
For a given location $j$ we define variable $x$ whose type is...

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

Let $T_i$ denote the set of locations within process $P_i$.

Let $\mathcal{P}$ be a program. We proceed to construct the...

State Variables

The state variables $V$ for system $\mathcal{P}$ consist of the data variables $Y$ which are declared at the head of the program and its processes, and the control variables $z_i$ ranges over the location set of each process $P_i$, for $i = 1, \ldots, m$. The value of $z_i$ in a state $s$ denotes the current location of control in the execution of process $P_i$.

$V = \{Y, z_1, \ldots, z_m\} = \Pi_i \{Y_i, z_i\}$

List of $\pi$.

State Variables

Hardware Verification, NYU, Fall, 2004

A. Pnueli

Lecture 1: Modeling Systems
The Initial Condition

Let \( \phi \) denote the data precondition of program \( P \). We define the initial condition

\[
\phi \land w_0 \land w_1 \land \cdots \land w_i = 1 \land 1
\]

as \( \Theta \) for \( \Theta \) denote the data precondition of program \( P \). We define the initial condition

\[
\phi_0 \land w_0 \land w_1 \land \cdots \land w_i = 1 \land 1
\]

where \( \phi_0 \) is the initial location of process \( P \). 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 \( \eta \), include the conjunct \( \theta \), where \( \theta \) denotes the empty list.

\[
\mathcal{V} = \alpha \land \alpha \land \mathcal{V}
\]

denotes the empty list.
and contributes to the requirement $\mathcal{L}$.

$$(\{\bar{y}, y\} - \Lambda) \pres \vee e = \bar{y} \vee \bar{y} \at \vee \bar{y}$$

The assignment statement

The considered statement.

Stating that all the variables in the variable set $\Lambda$ are preserved by the

$$(\bar{y} = \bar{y}) \vee (\Omega) \pres$$

We use the notation $\pres(\Omega)$ as an abbreviation for

belonging.

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

Transition Relation, Justice, and Compassion

A. Pnueli
The \texttt{await} statement `\begin{align*} j : \texttt{await} \ b; \end{align*}` \texttt{k : contributo}\texttt{s to the disjunct \begin{align*} j \at \ 0 \wedge b \at \ b \wedge \pres \ (V \ f \ i) \end{align*}} and contributes to the requirement \begin{align*} J \therefore \end{align*}.

\texttt{The \texttt{request} statement `\begin{align*} j : \texttt{request} \ r; \end{align*}` \texttt{k : contributo}\texttt{s to the disjunct \begin{align*} j \at \ r \wedge 0 < r \at \ r \wedge r \at \ r \wedge \pres \ (V \ f \ i); r \at \ r \at \ r \at \ r \end{align*}} and contributes to the requirement \begin{align*} C \therefore \end{align*}.

\texttt{The \texttt{release} statement `\begin{align*} j : \texttt{release} \ r; \end{align*}` \texttt{k : contributo}\texttt{s to the disjunct \begin{align*} j \at \ r \wedge 0 < r \at \ r \wedge r \at \ r \wedge \pres \ (V \ f \ i); r \at \ r \at \ r \at \ r \end{align*}} and contributes to the requirement \begin{align*} J \therefore \end{align*}.

which stays forever at \begin{align*} \forall \ q \wedge \at \ q \wedge \pres \ (V \ f \ i); q \at \ q \at \ q \at \ q \end{align*} disallowing an execution

\texttt{The \texttt{wait} statement `\begin{align*} j : \texttt{wait} \ q; \end{align*}` \texttt{k : contributo}\texttt{s to the disjunct \begin{align*} j \at \ q \wedge q \at \ q \wedge \pres \ (V \ f \ i); q \at \ q \at \ q \at \ q \end{align*}}
The statement \( \text{Non-Critical;} \) and contributes to the disjunct \( j \). In contrast to non-critical and contributes to the requirement - \( \exists \mathcal{F} \). This corresponds to the assumption that non-critical sections may fail to terminate.

\[
\left( \{ \exists \mathcal{F} \} - \Lambda \right) \text{press} \land \exists \mathcal{F} \land \exists \mathcal{F} \land \exists \mathcal{F}
\]

The statement \( \text{Critical;} \) contributes to the disjunct \( d \). In contrast to non-critical and does not contribute any fairness requirement.

\[
\left( \{ \exists \mathcal{F} \} - \Lambda \right) \text{press} \land \exists \mathcal{F} \land \exists \mathcal{F} \land \exists \mathcal{F}
\]

The statement \( \text{Non-Critical;} \) contributes to the disjunct \( d \). In contrast to non-critical and does not contribute any fairness requirement.
Compound Statements

- **The conditional statement** 
  
  \[
  \text{if } b \text{ then } S_1 \text{ else } S_2
  \]

  contributes to the disjunct

  \[
  \left( \{?\} - \Lambda \right) \text{pres} \lor \left( \begin{array}{c}
  \text{at-}\gamma \\
  \land \\
  \text{at-}\gamma
  \end{array} \right) \lor \gamma
  \]

- **The while statement** 
  
  \[
  \text{while } b \text{ do } S_1; S_2
  \]

  contributes to the disjunct

  \[
  \left( \{?\} - \Lambda \right) \text{pres} \lor \left( \begin{array}{c}
  \text{at-}\gamma \\
  \land \\
  \text{at-}\gamma
  \end{array} \right) \lor \gamma
  \]

- **The conditional statement** 
  
  \[
  \text{if } q \text{ then } S_1 \text{ else } S_2
  \]

  contributes to the disjunct

  \[
  \left( \{?\} - \Lambda \right) \text{pres} \lor \left( \begin{array}{c}
  \text{at-}\gamma \\
  \land \\
  \text{at-}\gamma
  \end{array} \right) \lor \gamma
  \]

- **The conditional statement** 
  
  \[
  \text{if } q \text{ then } S_1 \text{ else } S_2
  \]

  contributes to the disjunct

  \[
  \left( \{?\} - \Lambda \right) \text{pres} \lor \left( \begin{array}{c}
  \text{at-}\gamma \\
  \land \\
  \text{at-}\gamma
  \end{array} \right) \lor \gamma
  \]
Asynchronous Communication

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

- The asynchronous send statement \( \langle a \rangle \mu = \mu \land \forall \beta \neq \null \upon \beta - \langle a \rangle \) contributes to the disjunct \( d \) of the asynchronous communication requirement \( \langle a \rangle \) which is either an asynchronous channel or \( \infty \).

- The receive statement \( \langle a \rangle x = x \land \forall \beta \neq \null \upon \beta - \langle a \rangle \) also contributes to the disjunct \( d \) of the asynchronous communication requirement \( \langle a \rangle \) which is either an asynchronous channel or \( \infty \).

Note that the condition \( \langle a \rangle x = x \) is always true.

- The condition \( \forall \beta \neq \null \upon \beta - \langle a \rangle \) also contributes to the asynchronous communication requirement \( \langle a \rangle \) which is either an asynchronous channel or \( \infty \).

- The asynchronous communication requirement \( \langle a \rangle \) which is either an asynchronous channel or \( \infty \) is always true.

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

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

\[ (a_m \land \text{at}\, m, \quad \text{at} \quad \exists x: x = e) \land (a_m \land \text{at}\, m, \quad \text{at} \quad \exists x: x = e) \]

Such a pair also contributes to the requirement \( C \) such that \( \exists x: x = e \land (a_m \land \text{at}\, m, \quad \text{at} \quad \exists x: x = e) \)

\[ \{ \lbrack (n_m, (\exists x: x = e \land \text{at} \quad m, \quad \text{at} \quad (\exists x: x = e) - \Lambda) \lor \text{press} \} \land e \} \land \text{at} \quad m \land \text{at} \quad m \land \text{at} \quad m \land \text{at} \quad m \]

Contributes to the disjunction:

\[ \forall \exists \in C : \text{at} \quad e \Rightarrow \text{at} \quad e \]

Statements: let \( a \) be a synchronous channel. Each pair of matching send and receive

Synchronous Communication
The Idling Transition

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

\( \text{press} \ : \ \text{Id} \)
Consider the following circuit which implements a counter modulo 8.

\[ \neg f_i \oplus (\neg f_i \lor \neg o_i) = f_i \lor f_i \oplus o_i = \neg f_i \lor \neg o_i = o_i \]

By:

The state variables are given as:

\[ \{ f_i, o_i \} : \bigwedge \text{boolean} \]

Representations of Circuits as FDS's
Computation of the Circuit

\[ y_0' = y_0^2 \]

\[ y_1' = y_0 \]

\[ y_2' = (y_0 \land y_1) \lor \neg y_0 \]

We have computations such as:

\[ (1' \ominus (1 \lor 0) = 1') \lor (1 \ominus 0 = 0) \]

With the transition relation of the circuit.
A general sequential circuit has the following form:

\[ (f_i, x)^u f = u f_i \lor \cdots \lor (f_i, x)^{l} f = l f_i \quad : d \]

where \( f_1, \ldots, f_n \) are boolean functions expressing the outputs of the combinational circuit as a function of the inputs \( x_1, \ldots, x_m, y_1, \ldots, y_n \). The combinational circuit must be acyclic.
Consider another version of the modulo-8 counter.

The transition relation can be computed as:

\[
\begin{align*}
\bar{c}_2 \oplus (\bar{f}_1 \lor (\bar{f}_1 \lor \bar{x})) &= \bar{f}_2 \\
\lor \quad \bar{f}_1 \oplus (\emptyset \lor \bar{x}) &= f_1 \\
\lor \quad 0 \oplus 0x &= \emptyset
\end{align*}
\]

Computation of the Combinational Transfer Functions

A. Pnueli
Using Auxiliary Variables

\[ \forall (y_0 = x_0 \lor c_0 = \overline{c_0}) \lor (y_0 = \overline{x_0} \lor c_0 = 0) \lor (0 = x = 0 \lor c_0 = 0) \lor (0 = x = 0 \lor c_0 = 0) \]

Assigning an auxiliary variable to each gate output, we can also obtain the transition relation.
These are not really variables

The auxiliary variables $c_0, c_1$ can be viewed as names for subexpressions rather than as variables. Thus, the transition relation

\[
\begin{align*}
\forall (z_1 \oplus c_1 = c_1) \land (z_1 \oplus c_0 = c_1) \land (z_0 \lor 0x = 0c) \land (z_0 \lor 0x = 0d)
\end{align*}
\]

can also be written as

\[
\left( \begin{array}{c}
\forall (z_1 \oplus c_1 = c_1) \\
\forall (z_1 \oplus c_0 = c_1) \\
\forall (z_0 \lor 0x = 0c) \\
\forall (z_0 \lor 0x = 0d)
\end{array} \right)
\]

Note that the use of these names can reduce the size of the transition relation by up to an exponential factor.