Verification and Synthesis of Reactive Modules

Amir Pnueli

Weizmann Institute of Sciences and New York University

LASER Summer School, September, 2005

Including Joint work with:

Yonit Kesten  
Elad Shahar  
Oded Maler, E. Asarin, Joseph Sifakis  
Nir Piterman  

BGU  
Weizmann  
Verimag, Grenoble, France  
EPFL

Verification of Reactive Programs, Elba, Sept., 2005
Lectures Outline

- Overview of System Synthesis.
- Fair Discrete Systems and their Computations.
- Model Checking Invariance and feasibility.
- Temporal Testers and general LTL Model Checking.
- Controller Synthesis via Games.
- Synthesis from Recurrence Specifications.
- Synthesis from Reactivity Specifications. – The general case.
Motivation

Why verify, if we can automatically synthesize a program which is correct by construction?
A Brief History of System Synthesis

In 1965 Church formulated the following Church problem: Given a circuit interface specification (identification of input and output variables) and a behavioral specification,

- Determine if there exists an automaton (sequential circuit) which realizes the specification.
- If the specification is realizable, construct an implementing circuit.

The specification was given in the sequence calculus which is an explicit-time temporal logic.
Example of a Specification: Arbiter

The protocol for each client:
The Behavioral Specification

\[ \bigwedge_i \forall t : (r_i[t] = g_i[t] \rightarrow g_i[t + 1] = g_i[t]) \land (r_i[t] \neq g_i[t] \rightarrow r_i[t + 1] = r_i[t]) \] \land

\[ \bigwedge_{i \neq j} \forall t : \neg g_i[t] \lor \neg g_j[t] \] \land

\[ \bigwedge_i \forall t : r_i[t] \neq g_i[t] \rightarrow \exists s \geq t : r_i[s] = g_i[s] \]

Is this specification realizable?

The essence of synthesis is the conversion

From relations to Functions.
From Relations to Functions

Consider a computational program:

- The relation \( x^2 = y \) is a specification for the program computing the function \( y = \sqrt{x} \).
- The relation \( x \models y \) is a specification for the program that finds a satisfying assignment to the CNF boolean formula \( x \).

Checking is easier than computing.
Solutions to Church’s Problem

In 1969, M. Rabin provided a first solution to Church’s problem. Solution was based on automata on Infinite Trees. All the concepts involving $\omega$-automata were invented for this work.

At the same year, Büchi and Landweber provided another solution, based on infinite games.

These two techniques (Trees and Games) are still the main techniques for performing synthesis.
Synthesis of Reactive Modules from Temporal Specifications

Around 1981 Wolper and Emerson, each in his preferred brand of temporal logic (linear and branching, respectively), considered the problem of synthesis of reactive systems from temporal specifications.

Their (common) conclusion was that specification $\varphi$ is realizable iff it is satisfiable, and that an implementing program can be extracted from a satisfying model in the tableau. A typical solution they would obtain for the arbiter problem is:

Such solutions are acceptable only in circumstances when the environment fully cooperate with the system.
Next Step: Realizability $\neq$ Satisfiability

There are two different reasons why a specification may fail to be feasible.

Inconsistency

\[ \Diamond g \land \Box \neg g \]

Unrealizability  For a system

Realizing the specification

\[ g \leftrightarrow \Diamond r \]

requires clairvoyance.
A Synthesized Module Should Maintain Specification Against Adversarial Environment

In 1998, Rosner claimed that realizability should guarantee the specification against all possible (including adversarial) environment.

To solve the problem one must find a satisfying tree where the branching represents all possible inputs:

Can be formulated as satisfaction of the $\text{CTL}^*$ formula

$$\text{A} \varphi \land \text{A} \Box (\text{EX}(\overline{r_1} \land \overline{r_2}) \land \text{EX}(\overline{r_1} \land r_2) \land \text{EX}(r_1 \land \overline{r_2}) \land \text{EX}(r_1 \land r_2))$$
Bad Complexity

Rosner and P have shown [1989] that the synthesis process has worst case complexity which is doubly exponential. The first exponent comes from the translation of $\varphi$ into a non-deterministic Büchi automaton. The second exponent is due to the determinization of the automaton.

This result doomed synthesis to be considered highly untractable.
In 1989, Ramadge and Wonham introduced the notion of controller synthesis and showed that for a specification of the form $\square p$, the controller can be synthesized in linear time.

In 1998, Asarin, Maler, P, and Sifakis, extended controller synthesis to timed systems, and showed that for specifications of the form $\square p$ and $\Diamond q$, the problem can be solved by symbolic methods in linear time.
Lessons to be Learned from these Lectures

- Program (and design) synthesis is a tractable process.
- It can be solved using symbolic methods based on fixed-point iterations in a way very similar to model checking.
- The complexity of the solution is always polynomial where, unlike model checking, the degree of the polynomial depends on the structural complexity of the specification $\varphi$.
- For a very large class of specifications, arising in practice, the degree is $3$, i.e., the problem can be solved in time $n^3$. 
Lectures Outline

- Overview of System Synthesis.
- **Fair Discrete Systems and their Computations.**
- Model Checking Invariance and feasibility.
- **Temporal Testers** and general LTL Model Checking.
- Controller Synthesis via Games.
- Synthesis from Recurrence Specifications.
- Synthesis from Reactivity Specifications. – The general case.
As our computational model, we take fair discrete systems. An FDS $D = \langle V, \Theta, \rho, J, C \rangle$ consists of:

- $V$ – A finite set of typed state variables. A $V$-state $s$ is an interpretation of $V$. Denote by $\Sigma_V$ – the set of all $V$-states.

- $\Theta$ – An initial condition. A satisfiable assertion that characterizes the initial states.

- $\rho$ – A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (current) and primed (next) versions of the state variables. For example, $x' = x + 1$ corresponds to the assignment $x := x + 1$.

- $J = \{J_1, \ldots, J_k\}$ A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many $J_i$-states for each $J_i$, $i = 1, \ldots, k$.

- $C = \{\langle p_1, q_1 \rangle, \ldots, \langle p_n, q_n \rangle\}$ A set of compassion (strong fairness) requirements. Infinitely many $p_i$-states imply infinitely many $q_i$-states.
A Simple Programming Language: SPL

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

Example: Program ANY-Y

Consider the program

\[
\begin{align*}
x, y : \text{natural initially} & \quad x = y = 0 \\
\ell_0 : \quad & \text{while } x = 0 \text{ do} \\
\begin{align*}
[\ell_1 : \quad & y := y + 1] \\
\ell_2 : & \\
\end{align*} \\
\text{-- } P_1 \text{ --} \\
\end{align*}
\]

\[
\begin{align*}
\begin{align*}
[\ell_0 : \quad & \text{while } x = 0 \text{ do} \\
\ell_1 : & y := y + 1] \\
\end{align*} & \parallel \\
\begin{align*}
[\ell_2 : & \text{-- } P_2 \text{ --}] \\
\end{align*} \\
\end{align*}
\]

\[
\begin{align*}
\begin{align*}
[\ell_0 : \quad & \text{while } x = 0 \text{ do} \\
\ell_1 : & y := y + 1] \\
\ell_2 : & \text{-- } P_1 \text{ --} \\
\end{align*} & \parallel \\
\begin{align*}
[\ell_0 : \quad & \text{while } x = 0 \text{ do} \\
\ell_1 : & y := y + 1] \\
\ell_2 : & \text{-- } P_2 \text{ --} \\
\end{align*} \\
\end{align*}
\]
The Corresponding FDS

- **State Variables** \( V: \)
  
  \[
  \begin{align*}
  x, y & : \text{natural} \\
  \pi_1 & : \{ \ell_0, \ell_1, \ell_2 \} \\
  \pi_2 & : \{ m_0, m_1 \}
  \end{align*}
  \]

- **Initial condition**: \( \Theta: \pi_1 = \ell_0 \land \pi_2 = m_0 \land x = y = 0. \)

- **Transition Relation**: \( \rho: \rho_I \lor \rho_{\ell_0} \lor \rho_{\ell_1} \lor \rho_{m_0}, \) with appropriate disjunct (transition) for each statement. For example, the disjuncts \( \rho_I \) and \( \rho_{\ell_0} \) are
  
  \[
  \rho_I : \quad \pi_1' = \pi_1 \land \pi_2' = \pi_2 \land x' = x \land y' = y
  \]

  \[
  \rho_{\ell_0} : \quad \pi_1 = \ell_0 \land \left( x = 0 \land \pi_1' = \ell_1 \lor x \neq 0 \land \pi_1' = \ell_2 \right) \land \pi_2' = \pi_2 \land x' = x \land y' = y
  \]

- **Justice set**: \( \mathcal{J}: \{ \neg \text{at}_{-\ell_0}, \neg \text{at}_{-\ell_1}, \neg \text{at}_{-m_0} \}. \) Usually, we have a justice transition expressing the disableness of each just transition.

- **Compassion set**: \( \mathcal{C}: \emptyset. \)
**Computation**

Let $D$ be an FDS for which the above components have been identified. The state $s'$ is defined to be a $D$-successor of state $s$ if

$$\langle s, s' \rangle \models \rho_D(V, V').$$

We define a computation of $D$ to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, \ldots,$$

satisfying the following requirements:

- **Initiality:** $s_0$ is initial, i.e., $s_0 \models \Theta$.

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

- **Justice:** For each $J \in \mathcal{J}$, $\sigma$ contains infinitely many $J$-positions. This guarantees that every just transition is disabled infinitely many times.

- **Compassion:** For each $\langle p, q \rangle \in \mathcal{C}$, if $\sigma$ contains infinitely many $p$-positions, it must also contain infinitely many $q$-positions. This guarantees that every compassionate transition which is enabled infinitely many times is also taken infinitely many times.
Examples of Computations

Identification of the FDS $D_P$ corresponding to a program $P$ gives rise to a set of computations $\text{Comp}(P) = \text{Comp}(D_P)$.

The following computation of program ANY-Y corresponds to the case that $m_0$ is the first executed statement:

\[
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\ell_0} \\
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 0 \rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots
\]

The following computation corresponds to the case that statement $\ell_1$ is executed before $m_0$.

\[
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi_1: \ell_1, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: \ell_0, \pi_2: m_0; x: 0, y: 1 \rangle \xrightarrow{m_0} \langle \pi_1: \ell_0, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\ell_0} \\
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: 1 \rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots
\]

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement $\ell_0$ $n$ times and then terminates in the final state

\[
\langle \pi_1: \ell_2, \pi_2: m_1; x: 1, y: n \rangle.
\]
A Non-Computation

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

Thus, the sequence

\[
\langle \pi_1: l_0, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0; x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0; x: 0, y: 1 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0; x: 0, y: 1 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0; x: 0, y: 2 \rangle \xrightarrow{\ell_0} \langle \pi_1: l_1, \pi_2: m_0; x: 0, y: 2 \rangle \xrightarrow{\ell_1} \\
\langle \pi_1: l_0, \pi_2: m_0; x: 0, y: 3 \rangle \xrightarrow{\ell_0} \ldots
\]

in which statement \( m_0 \) is never executed is not an admissible computation. This is because it violates the justice requirement \( \neg \text{at} \ m_0 \) contributed by statement \( m_0 \), by having no states in which this requirement holds.

This illustrates how the requirement of justice ensures that program ANY-Y always terminates.

Justice guarantees that every (enabled) process eventually progresses, in spite of the representation of concurrency by interleaving.
Justice is not Enough. You also Need Compassion

The following program $\text{MUX-SEM}$, implements mutual exclusion by semaphores.

$$y : \text{natural initially } y = 1$$

$$P_1 :: \begin{bmatrix} \ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-critical} \\
\ell_2 : \text{request } y \\
\ell_3 : \text{Critical} \\
\ell_4 : \text{release } y \end{bmatrix} \parallel P_2 :: \begin{bmatrix} m_0 : \text{loop forever do} \\
m_1 : \text{Non-critical} \\
m_2 : \text{request } y \\
m_3 : \text{Critical} \\
m_4 : \text{release } y \end{bmatrix}$$

The compassion set of this program consists of

$$C : \{(\text{at}_\ell \ell_2 \land y > 0, \text{at}_\ell \ell_3), \ (\text{at}_m m_2 \land y > 0, \text{at}_m m_3)\}.$$ 

Usually, with a compassionate transition $\tau$, we associate the compassion requirement

$$(En(\tau), \ \text{taken}(\tau))$$
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 it’s critical section at $\ell_3$. Similar requirement for $P_2$.

Consider the state sequence:

\[
\begin{align*}
\sigma : \langle \ell_0, m_0, 1 \rangle &\rightarrow \cdots \rightarrow \langle \ell_2, m_2, 1 \rangle \mbox{ } m_2 \\
\langle \ell_2, m_3, 0 \rangle &\xrightarrow{m_3} \langle \ell_2, m_4, 0 \rangle \xrightarrow{m_4} \langle \ell_2, m_2, 1 \rangle \mbox{ } m_2 \\
\langle \ell_2, m_0, 1 \rangle &\xrightarrow{m_0} \langle \ell_2, m_1, 1 \rangle \xrightarrow{m_1} \langle \ell_2, m_2, 1 \rangle \mbox{ } m_2 \\
\langle \ell_2, m_3, 0 \rangle &\rightarrow \cdots ,
\end{align*}
\]

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 !!!
FDS Operations: Asynchronous Parallel Composition

The asynchronous parallel composition of systems $D_1$ and $D_2$, denoted by $D_1 \parallel D_2$, is given by $D = \langle V, \Theta, \rho, J, C \rangle$, where

\[
\begin{align*}
V &= V_1 \cup V_2 \\
\Theta &= \Theta_1 \land \Theta_2 \\
\rho &= (\rho_1 \land \text{pres}(V_2 - V_1)) \lor (\rho_2 \land \text{pres}(V_1 - V_2)) \\
J &= J_1 \cup J_2 \\
C &= C_1 \cup C_2
\end{align*}
\]

The predicate $\text{pres}(U)$ stands for the assertion $U' = U$, implying that all the variables in $U$ are preserved by the transition.

Asynchronous parallel composition represents the interleaving-based concurrency which is assumed in shared-variables models.

**Claim 1.** $D(P_1 \parallel P_2) \sim D(P_1) \parallel D(P_2)$
Synchronous Parallel Composition

The synchronous parallel composition of systems $D_1$ and $D_2$, denoted by $D_1 \parallel D_2$, is given by the FDS $D = \langle V, \Theta, \rho, J, C \rangle$, where

- $V = V_1 \cup V_2$
- $\Theta = \Theta_1 \land \Theta_2$
- $\rho = \rho_1 \land \rho_2$
- $J = J_1 \cup J_2$
- $C = C_1 \cup C_2$

Synchronous parallel composition is used for the construction of an observer: a system $O$ which observes and evaluates the behavior of an observed system $D$. Running $D \parallel O$, we let $D$ behave as usual, while $O$ observes its behavior.
Feasibility and Viability of Systems

An FDS $D$ is said to be **feasible** if $D$ has at least one computation.

A finite or infinite sequence of states is defined to be a **run** of an FDS $D$ if it satisfies the requirements of **initiality** and **consecution** but not necessarily any of the **fairness** requirements.

The FDS $D$ is defined to be **viable** if any finite run of $D$ can be extended to a computation of $D$.

**Claim 2.** *Every FDS derived from an SPL program is viable.*

Note that if $D$ is a viable system, such that its initial condition $\Theta_D$ is satisfiable, then $D$ is feasible.
Requirement Specification Language: Temporal Logic

Assume an underlying (first-order) assertion language. The predicate $\text{at}_{l_i}$ abbreviates the formula $\pi_j = l_i$, where $l_i$ is a location within process $P_j$.

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

- $\bigcirc$ – Next
- $\bigotimes$ – Previous
- $\mathcal{U}$ – Until
- $S$ – Since

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

- $\Diamond p = 1 \mathcal{U} p$ – Eventually
- $\square p = \neg \Diamond \neg p$ – Henceforth
- $p \mathcal{W} q = \square p \lor (p \mathcal{U} q)$ – Waiting-for, Unless, Weak Until
- $\Diamond p = 1 S p$ – Sometimes in the past
- $\square p = \neg \Diamond \neg p$ – Always in the past
- $p \mathcal{B} q = \square p \lor (p S q)$ – Back-to, Weak Since

A model for a temporal formula $p$ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state $s_j$ provides an interpretation for the variables of $p$. 
Semantics of LTL

Given a model $\sigma$, we define the notion of a temporal formula $p$ holding at a position $j \geq 0$ in $\sigma$, denoted by $(\sigma,j) \models p$:

- For an assertion $p$,
  
  $$(\sigma,j) \models p \iff s_j \models p$$

  That is, we evaluate $p$ locally on state $s_j$.

- $(\sigma,j) \models \neg p \iff (\sigma,j) \not\models p$

- $(\sigma,j) \models p \lor q \iff (\sigma,j) \models p \text{ or } (\sigma,j) \models q$

- $(\sigma,j) \models \Diamond p \iff (\sigma,j+1) \models p$

- $(\sigma,j) \models p \mathcal{U} q \iff$ for some $k \geq j$, $(\sigma,k) \models q$, and for every $i$ such that $j \leq i < k$, $(\sigma,i) \models p$

- $(\sigma,j) \models \Box p \iff j > 0$ and $(\sigma,j-1) \models p$

- $(\sigma,j) \models p \mathcal{S} q \iff$ for some $k \leq j$, $(\sigma,k) \models q$, and for every $i$ such that $j \geq i > k$, $(\sigma,i) \models p$

For a model $\sigma$, we say that $\sigma$ satisfies $\varphi$, written $\sigma \models \varphi$, if $(\sigma,0) \models \varphi$. For an FDS $D$ and an LTL formula $\varphi$, we say that $\varphi$ is $D$-valid, denoted $D \models \varphi$, if all computations of $D$ satisfy $\varphi$. 
Reading Exercises

Following are some temporal formulas $\varphi$ and what do they say about a state sequence $\sigma : s_0, s_1, \ldots$ such that $\sigma \models \varphi$:

- $p \rightarrow \Diamond q$ — If $p$ holds at $s_0$, then $q$ holds at $s_j$ for some $j \geq 0$.

- $\square (p \rightarrow \Diamond q)$ — Every $p$ is followed by a $q$. Can also be written as $p \Rightarrow \Diamond q$.

- $\square \Diamond q$ — The sequence $\sigma$ contains infinitely many $q$’s.

- $\Diamond \square q$ — All but finitely many states in $\sigma$ satisfy $q$. Property $q$ eventually stabilizes.

- $q \Rightarrow \Diamond p$ — Every $q$ is preceded by a $p$ — causality.

- $(\neg r) W q$ — $q$ precedes $r$. $r$ cannot occur before $q$ — precedence. Note that $q$ is not guaranteed, but $r$ cannot happen without a preceding $q$.

- $(\neg r) W (q \land \neg r)$ — $q$ strongly precedes $r$.

- $p \Rightarrow (\neg r) W q$ — Following every $p$, $q$ precedes $r$. 
Temporal Specification of Properties

A \( \mathcal{D} \)-valid formula \( \varphi \) specifies a property of \( \mathcal{D} \).

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

- **Mutual Exclusion** – No computation of the program includes a state in which process \( P_1 \) is at \( \ell_3 \) while \( P_2 \) is at \( m_3 \). Specifiable by

  \[ \Box \neg (at_{\ell_3} \land at_{m_3}) \]

- **Accessibility** for \( P_1 \) – Whenever process \( P_1 \) is at \( \ell_2 \), it shall eventually reach its critical section at \( \ell_3 \). Specifiable by

  \[ \Box (at_{\ell_2} \implies \Diamond at_{\ell_3}) \]

  or, equivalently

  \[ (at_{\ell_2} \implies \Diamond at_{\ell_3}) \]
Lectures Outline

- Overview of System Synthesis.
- Fair Discrete Systems and their Computations.
- Model Checking Invariance and feasibility.
- Temporal Testers and general LTL Model Checking.
- Controller Synthesis via Games.
- Synthesis from Recurrence Specifications.
- Synthesis from Reactivity Specifications. – The general case.
Checking whether an FDS is Feasible

An FDS is called feasible if it has at least one computation. We will present an algorithm for checking whether a given finite-state FDS is feasible.

Obviously, all FDS’s derived from an SPL program are feasible. We intend to use feasibility checking for verification, based on the equivalence

\[ \mathcal{D} \models \varphi \quad \text{iff} \quad \mathcal{D} \parallel Tester[\neg \varphi] \text{ is infeasible}, \]

where Tester[\neg \varphi] is a special FDS, whose computations are all the sequences satisfying the temporal property \( \neg \varphi \). Infeasibility of \( \mathcal{D} \parallel Tester[\neg \varphi] \) means that \( \mathcal{D} \) has no computation violating \( \varphi \), and therefore, all computations of \( \mathcal{D} \) satisfy \( \varphi \).

A run of \( \mathcal{D} \) is a finite or infinite state sequence which satisfies the requirements of initiality and consecution but is not necessarily fair.

A run segment is a finite state sequence which satisfies the requirement of consecution.

A state \( s \) is \( \mathcal{D} \)-accessible if it appears in some \( \mathcal{D} \)-run.
The State-Transition Graph

For a given FDS $\mathcal{D}$, we define the state-transition graph to be a directed graph whose nodes are all the $\mathcal{D}$-accessible states, and whose edges connect node $s$ to node $s'$ iff $s'$ is a $\mathcal{D}$-successor of $s$. 
Example: a Simpler MUX-SEM

Below, we present a simpler version of program MUX-SEM.

\[ y: \text{natural initially } y = 1 \]

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

\[ \langle \text{when } y = 1 \text{ do } y := 0 \rangle \text{ and } y := 1. \]
**Observation 1.** *Every computation of $D$ corresponds to an infinite initialized path in $G_D$.\*  

However, not every infinite initialized $G_D$-path corresponds to a $D$-computation.*
From Combinations to Fair Subgraphs

A subgraph $S \subseteq G_D$ is called a **perpetual** if every state $s \in S$ has an $S$-successor. For example, $\{\langle N_1, N_2, 1 \rangle, \langle T_1, N_2, 1 \rangle, \langle C_1, N_2, 0 \rangle \}$, and $\{\langle N_1, N_2, 1 \rangle\}$ are both perpetual subgraphs of the state-transition graph of MUX-SEM.

A subgraph $S$ is called **just** if every $s \in S$ initiates an $S$-path leading to a $J$-state for every justice requirement $J \in J$.

The subgraph $S$ is called **compassionate** if, for every compassion requirement $(p, q) \in C$, either $S$ contains no $p$-states, or every $s \in S$ initiates an $S$-path leading to a $q$-state.

A subgraph $S$ is **fair** if it is a perpetual, just, and compassionate.

Let $\pi$ be an infinite initialized path in $G_D$. We denote by $\text{Inf}(\pi)$ the set of states which appear infinitely many times in $\pi$. 
A necessary and Sufficient Condition

The following claim relates computations of $\mathcal{D}$ to fair subgraphs of $G_D$.

**Claim 3.** The infinite initialized $G_D$-path $\pi$ is a computation of $\mathcal{D}$ iff $\text{Inf}(\pi)$ is a fair subgraph of $G_D$.

**Corollary 4.** A system $\mathcal{D}$ is feasible iff $G_D$ contains a fair subgraph.
Symbolic Finite-State Verification

In symbolic analysis, we use assertions to represent sets of states. Thus, $\Theta$ represents the set of all initial states. For finite-state systems, we can always use boolean assertions to encode sets of states.

Define the predecessor predicate transformer:

$$\rho \circ \psi = \exists V' : \rho(V, V') \land \psi(V')$$

Obviously, for an assertion $\psi$,

$$s \models \rho \circ \psi \iff \text{some } \rho\text{-successor of } s \text{ satisfies } \psi.$$

The immediate predecessor transformer can be iterated to yield the eventual predecessor transformer:

$$\rho^* \circ \psi = \psi \lor \rho \circ \psi \lor \rho \circ (\rho \circ \psi) \lor \rho \circ (\rho \circ (\rho \circ \psi)) \lor \cdots$$
Formulation as Fixed Points

Consider a recursive equation of the general form $y = f(y)$, where $y$ is an assertion representing a set of states. Such an equation is called a fix-point equation.

Not every fix-point equation has a solution. For example, the equation $y = \neg y$ has no solution.

The assertional expression $f(y)$ is called monotonic if it satisfies the requirement

$$\|y_1\| \subseteq \|y_2\| \text{ implies } \|f(y_1)\| \subseteq \|f(y_2)\|$$
Solutions to Fix-point Equations

Every assertional expression \( f(y) \) which is constructed out of the assertion variable \( y \) and arbitrary constant assertions, to which we apply the boolean operators \( \lor \) and \( \land \), and the predecessor operator \( \rho \diamond p \) is monotonic.

Consider a fix-point equation

\[
y = f(y)
\]  

(1)

It may have 0, one, or many solutions. For example, the equation \( y = y \) has many solutions. A solution \( y_m \) is called a minimal solution if it satisfies \( \|y_m\| \subseteq \|y\| \) for any solution \( y \) of Equation (1). A solution \( y_M \) is called a maximal solution if it satisfies \( \|y_M\| \supseteq \|y\| \) for any solution \( y \) of Equation (1). We denote by \( \mu y.f(y) \) and \( \nu y.f(y) \) the minimal and maximal solutions, respectively.

Claim 5. If \( f(y) \) is a monotonic expression, then the fix-point equation \( y = f(y) \) has both a minimal and a maximal solution which can be obtained by the iteration sequence

\[
y_1 = f(y_0), \ y_2 = f(y_1), \ y_3 = f(y_2), \ldots
\]

where \( y_0 = 0 \) for the minimal solution, and \( y_0 = 1 \) for the maximal solution.
Expressing the Eventual Predecessor

A generalized version of the eventual predecessor can be expressed by a minimal fix-point expression:

\[(p \land \rho_D)^* \diamond q = \mu y. q \lor p \land \rho_D \diamond y\]

This is because the fix-point expression generates the following approximation sequence:

\[
\begin{align*}
y_0 &= 0 \\
y_1 &= q \lor 0 = q \\
y_2 &= q \lor p \land \rho_D \diamond y_1 = q \lor p \land \rho_D \diamond q \\
y_3 &= q \lor p \land \rho_D \diamond y_2 = q \lor p \land \rho_D \diamond q \lor p \land \rho_D \diamond (p \land \rho_D \diamond q) \\
&\quad \ldots
\end{align*}
\]

Characterizing the set of all states which initiate a \(p\)-path leading to a \(q\)-state.
A Symbolic Algorithm for Model Checking Invariance

Algorithm $\text{INV}(D, p) : \text{assertion}$ — Check that FDS $D$ satisfies $\text{Inv}(p)$, using symbolic operations

$$
\begin{align*}
\text{new} & : \text{assertion} \\
1. \quad \text{new} & := \neg p \\
2. \quad \text{Fix} (\text{new}) \text{ do} \\
3. \quad \text{new} & := \text{new} \lor (\rho_D \diamond \text{new}) \\
4. \quad \text{return} \quad \Theta_D \land \text{new}
\end{align*}
$$

where

$$
\text{Fix} (y) \text{ do } S \quad = \quad \text{old} := \neg y; \text{ While } (y \neq \text{old}) \text{ do } [\text{old} := y; S]
$$

The algorithm returns an assertion characterizing all the initial states from which there exists a finite path leading to violation of $p$. It returns the empty (false) assertion iff $D$ satisfies $\text{Inv}(p)$.

An equivalent formulation is

$$
\text{return} \quad \Theta_D \land \mu y : \neg p \lor \rho_D \diamond y
$$
We iterate as follows:

\[ \varphi_0 : \quad \pi_1 = C \land \pi_2 = C \]

\[ \varphi_1 : \quad \varphi_0 \lor \begin{cases} \pi_1 = T \land y = 1 \land \pi'_1 = C \land y' = 0 \\ \pi_2 = T \land y = 1 \land \pi'_2 = C \land y' = 0 \end{cases} \sim (\pi_1 = \pi_2 = C) \]

\[ \pi_1 = \pi_2 = C \lor \pi_1 = T \land \pi_2 = C \land y = 1 \land \pi_1 = C \land \pi_2 = T \land y = 1 \]

\[ \varphi_2 : \quad \varphi_1 \lor \pi_1 = N \land \pi_2 = C \land y = 1 \lor \pi_1 = C \land \pi_2 = N \land y = 1 \]

\[ \varphi_3 : \quad \varphi_2 \lor \pi_1 = C \land \pi_2 = C \land y = 0 \sim \varphi_2 \]

The last equivalence is due to the general property \( p \lor (p \land q) \sim p \).

If we intersect \( \varphi_3 \) with the initial condition \( \Theta : \pi_1 = N \land \pi_2 = N \land y = 1 \) we obtain 0 (false). We conclude that MUX-SEM satisfies \( \text{Inv}(-\left(\pi_1 = C \land \pi_2 = C\right)) \).
Symbolic Exploration Progresses in Layers

\[ \phi_0 \rightarrow \phi_1 \rightarrow \phi_2 \]

- \( \phi_0 \): \( C_1, C_2, - \)
- \( \phi_1 \):
  - \( T_1, C_2, 1 \)
  - \( N_1, C_2, 1 \)
- \( \phi_2 \):
  - \( C_1, T_2, 1 \)
  - \( C_1, N_2, 1 \)
In Search of Fair Subgraphs

Let $\varphi$ characterize a fair subgraph. Then we have:

**F1.** Each state $s \in S$ has a $\rho$-successor in $S$.

$$\varphi \rightarrow \rho_D \varphi$$

**F2.** For every state $s \in S$ and every justice requirement $J \in \mathcal{J}$, there exists an $S$-path leading from $s$ to some $J$-state.

$$\varphi \rightarrow (\varphi \land \rho_D)^* \lozenge (\varphi \land J)$$

**F3.** For every state $s \in S$ and every compassion requirement $(r, q) \in \mathcal{C}$, either there exists an $S$-path leading from $s$ to some $q$-state, or $s$ satisfies $\neg r$.

$$\varphi \rightarrow \neg p \lor (\varphi \land \rho_D)^* \lozenge (\varphi \land q)$$

We conclude:

$$\varphi \rightarrow \left\{ \begin{array}{l}
\rho \lozenge \psi \land \bigwedge_{J \in \mathcal{J}} (\rho \land \psi)^* \lozenge (J \land \psi) \\
\land \bigwedge_{(r,q) \in \mathcal{C}} [\neg p \lor (\rho \land \psi)^* \lozenge (q \land \psi)]
\end{array} \right\}$$
A Symbolic Algorithm for Checking Feasibility

Algorithm SET-FEASIBLE ($D$) : assertion — Compute the set of states which initiate a fair run, using symbolic operations.

1. $new := 1$
2. Fix ($new$) do
   begin
3. $new := new \land (\rho_D \diamond new)$
4. for each $J \in J$ do
5. $new := new \land ((new \land \rho_D)^* \diamond (new \land J))$
6. for each $(p,q) \in C$ do
7. $new := \begin{cases} new \land \neg p \\ \lor new \land ((new \land \rho_D)^* \diamond (new \land q)) \end{cases}$
   end
8. return ($\rho_D^* \diamond new$)
Correctness of the Algorithm

Claim 6. Algorithm SET-FEASIBLE terminates, with state \( s \) satisfying SET-FEASIBLE\((D)\) iff there exists a \( G_D \)-path leading from \( s \) to a fair subgraph of \( G_D \).

The proof is partitioned into three parts:

1. The Algorithm terminates: We define an ordering relation on assertions by letting

\[ p \preceq q \iff \|p\| \subseteq \|q\|. \]

Denote by \( new_j^i \) the assertion which is the (symbolic) value of variable \( new \) at the \( j \)th visit to line \( i \) (before executing line \( i \)).

Since all operations applied to variable \( new \) are of the form \( new \wedge E \) or a disjunction of such expressions, it is easy to see that lines 5, 7, and 9 only remove states from \( new \). Therefore, we have that \( new_j^{i+1} \preceq new_j^i = old_j^{i+1} \) for all \( j = 1, 2, \ldots \).

Since \( G_D \) is finite, the algorithm must terminate.
Correctness of the Algorithm: Completeness

Next, we prove that Algorithm SET-FEASIBLE is complete. Namely, if $S$ is a fair subgraph of $G_D$ and $s$ is a state leading to $S$, then $s \in \|\text{SET-FEASIBLE}(D)\|$.

To do so, we show that $S \subseteq \|new_{10}\|$ from which the claim of completeness follows.

The above inclusion follows by induction on the number of steps performed by the algorithm, where the induction basis is provided by

$$S \subseteq G_D = \|1\| = \|new_{1}\|,$$

and the induction step is supported by the fact that, due to $S$ being a fair subgraph, $S \subseteq \|new\|$ implies the following:

$$S \subseteq \|new \wedge (\rho_D \bowtie new)\|$$
$$S \subseteq \|new \wedge ((new \wedge \rho_D)^* \bowtie (new \wedge J))\| \quad \text{For every } J \in \mathcal{J}$$
$$S \subseteq \bigg\| \bigg( new \wedge \neg p \bigg\} \quad \text{For every } (p, q) \in \mathcal{C}$$
**Algorithm Correctness: Soundness**

Finally, we show that the algorithm is sound. Namely, if $s \in \|\text{SET-FEASIBLE}(D)\|$ then there exists $S$, a fair subgraph of $G_D$, and a path leading from $s$ to $S$.

When the algorithm terminates, we know that

1. Every $s \in \|\text{new}_{10}\|$ has a successor $s' \in \|\text{new}_{10}\|$.
2. Every $s \in \|\text{new}_{10}\|$ initiates a $\|\text{new}_{10}\|$-path leading to a $J \land \text{new}_{10}$-state, for every $J \in \mathcal{J}$.
3. Every $s \in \|\text{new}_{10}\|$ initiates a $\|\text{new}_{10}\|$-path leading to a $q \land \text{new}_{10}$-state or satisfies $\neg p$, for every $(p, q) \in \mathcal{C}$.

Assume that $s \in \|\text{SET-FEASIBLE}(D)\|$. Line 10 implies that $s$ is connected by a path $\pi$ to an $\|\text{new}_{10}\|$-state. Repeat the following successive extension of $\pi$ ad-infinitum, denoting the last state of $\pi$ by $s_\ell$:

1. Extend $\pi$ by a $\|\text{new}_{10}\|$-successor of $s_\ell$, guaranteed by 1.
2. For every $J \in \mathcal{J}$, extend $\pi$ by a $\|\text{new}_{10}\|$-path leading to a $J \land \text{new}_{10}$-state, guaranteed by 2.
3. For every $(p, q) \in \mathcal{C}$, if there exists a $\|\text{new}_{10}\|$-path $\pi'$ connecting $s_\ell$ to a $q \land \text{new}_{10}$-state, then extend $\pi$ by $\pi'$. Otherwise, do not extend $\pi$. When done, go to 1.

Can show that $S = \text{Inf}(\pi)$ is an $s$-reachable fair subgraph.
Expressed by Fix-points

Algorithm SET-FEASIBLE can be succinctly represented by the feasibility predicate of an FDS \( D \), given by the following fix-point formula:

\[
\text{FEASIBLE}(D) = \rho^* \nu \psi : \begin{cases} \\
\rho \diamond \psi \land \bigwedge_{J \in \mathcal{J}} (\rho \land \psi)^* \diamond (J \land \psi) \\
\land \bigwedge_{(p,q) \in C} \left[ \neg p \lor (\rho \land \psi)^* \diamond (q \land \psi) \right] \end{cases}
\]

\text{FEASIBLE}(D) characterizes all the states from which a fair subgraph of \( G_D \) is reachable.

The FDS \( D \) is feasible iff \( \text{FEASIBLE}(D) \land \Theta \) is satisfiable.
Example

As an example, consider the following FDS to which we apply SET-FEASIBLE:

\[ x : 0 \rightarrow x : 1 \rightarrow x : 2 \rightarrow x : 3 \rightarrow x : 4 \rightarrow x : 5 \]

with the fairness requirements:

\[ J_1 : x \neq 1 \]
\[ C_1 : (x = 3, x = 5) \]
\[ C_2 : (x = 2, x = 1) \]

We set \( \varphi_0 : \{0..5\} \) and then proceed as follows:

- Removing from \( \varphi_0 \) all \((x = 2)\)-states which do not have a \( \varphi_0 \)-path leading to an \((x = 1)\)-state, we are left with \( \varphi_1 : \{0, 1, 3, 4, 5\} \).
- Successively removing from \( \varphi_1 \) all states without successors, leaves \( \varphi_2 : \{3, 4\} \).
- Removing from \( \varphi_2 \) all \((x = 3)\)-states which do not have a \( \varphi_2 \)-path leading to a \((x = 5)\)-state, we are left with \( \varphi_3 : \{4\} \).
- No reasons to remove any further states from \( \varphi_3 : \{4\} \), so this is our final set.

Retaining all states from which \( \varphi_3 \) is reachable, we get SET-FEASIBLE = 1.
Temporal Testers

For every LTL formula \( \varphi \), there exists an FDS \( T[\varphi] \) called the temporal tester for \( \varphi \). This tester has a distinguished boolean variable \( x \), such that, in every \( \sigma \), a computation of \( T[\varphi] \) and every position \( j \geq 0 \), \( x[s_j] = 1 \) iff \( (\sigma, j) \models \varphi \).

An LTL formula whose principal operator is temporal, and such that it does not contain any nested temporal operator is called a basic formula.

We will only present testers for basic path formulas.
Example: a Tester for $\Diamond p$

\[
T[\Diamond p] : \left\{
\begin{array}{l}
V : \text{Vars}(p) \cup \{x\} \\
\Theta : 1 \\
\rho : x = p \lor x' \\
J : p \lor \neg x \\
C : \emptyset
\end{array}
\right.
\]

The justice requirement demands that either $p = 1$ infinitely many times, or $x = 0$ infinitely many times. This rules out a computation in which $p = 0$ and $x = 1$ continuously, even though such a state sequence satisfies the requirements of initiality and consecution.
Testers for $\Box p$ and $\neg \Box p$

$T[\Box p] : \begin{cases} 
V = \emptyset : Vars(p) \cup \{x\} \\
\Theta : 1 \\
\rho : x = p' \\
J = C : \emptyset 
\end{cases}$

$T[\neg \Box p] : \begin{cases} 
V = \emptyset : Vars(p) \cup \{x\} \\
\Theta : x = 0 \\
\rho : x' = p \\
J = C : \emptyset 
\end{cases}$

Note that $x = 0$ at the first position, since $\neg \Box p$ is always false at position 0.
Testers for $p \mathcal{U} q$, and $p \mathcal{S} q$

\[
T[p \mathcal{U} q] : \begin{cases} 
V = \emptyset : & Vars(p, q) \cup \{x\} \\
\Theta : & 1 \\
\rho : & x = q \lor (p \land x') \\
J : & q \lor \neg x \\
C : & \emptyset 
\end{cases}
\]

Note the justice requirement by which either $q$ or $x = 0$ should hold infinitely many times.

\[
T[p \mathcal{S} q] : \begin{cases} 
V = \emptyset : & Vars(p, q) \cup \{x\} \\
\Theta : & x = q \\
\rho : & x' = q' \lor (p' \land x) \\
J = C : & \emptyset 
\end{cases}
\]

Note that the initial value of $x$, representing $p \mathcal{S} q$, equals the value of $q$ at position 0.
Testers for Arbitrary LTL Formulas

Up to now we only considered testers for basic path formulas. We proceed to show how to construct a tester for an arbitrary LTL formula.

Let $f(\varphi)$ be a principally temporal LTL formula containing one or more occurrences of the basic formula $\varphi$. We denote by $f(x)$ the formula obtained from $f$ by replacing all occurrences of $\varphi$ by the boolean variable $x$. Then the construction principle is presented by the following recursive reduction formula:

$$T[f] = T[f(x\varphi)] \parallel T[\varphi]$$  \hspace{1cm} (2)

That is, we conjoin the tester for $\varphi$ to the recursively constructed tester for the simpler formula $f(x\varphi)$. 
Example

We illustrate this construction on the LTL formula $\square \Diamond p$ for an assertion $p$. Application of the reduction principle leads to

$$T[\square \Diamond p] = T[\square x\Diamond] \parallel T[\Diamond p]$$

Computing $T[\Diamond p]$ and $T[\square x\Diamond]$ separately and forming their synchronous parallel composition yields the following tester whose output variable is $x\Diamond$.

$$T[\square \Diamond p] : \begin{cases}
    V : & \text{Vars}(p) \cup \{x\Diamond, x\Box\} \\
    \Theta : & 1 \\
    \rho : & (x\Diamond = p \lor x'\Diamond) \land (x\Box = x\Diamond \land x'\Box) \\
    J : & \{\neg x\Diamond \lor p, x\Box \lor \neg x\Diamond\} \\
    C : & \emptyset
\end{cases}$$

In general, for a principally temporal formula $\psi$, $T[\psi] = T_1 \parallel \cdots \parallel T_k$, where $T_1, \ldots, T_k$ are the temporal testers constructed for the principally temporal subformulas of $\psi$. $T[\psi]$ contains $k$ auxiliary boolean variables, and the output variable of $T[\psi]$ is the output variable of $T_1$ — the last constructed tester.