Verification and Synthesis of Reactive Programs

Amir Pnueli

Weizmann Institute of Sciences and New York University

Mini-Course, Universita' di Roma La Sapienza June, 2006

Including Joint work with:

Yonit Kesten	BGU
Elad Shahar	Weizmann
Oded Maler, E. Asarin, Joseph Sifakis	Verimag, Grenoble, France
Nir Piterman	EPFL

Lectures Outline

- Overview of System Synthesis.
- Fair Discrete Systems and their Computations.
- Model Checking Invariance and response.
- 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?

Applying Mathematics to the Programming Problem

The mathematical paradigm considers a constraint $C(x)$, e.g.

$$2 < x \leq 10$$

and asks questions such as:

1. Does $x = 5$ satisfy the constraint?
2. Is the constraint satisfiable by some x?
3. Find an x which satisfies the constraint.
4. Find the best, say maximal, x which satisfies C.

Question: If x is the program, what is C?

Answer: C is the specification which the program should satisfy.

Program Verification solves Problem no. 1.

Program Synthesis solves Problems no. 2 and 3.

> Why perform a post-facto Verification if you can synthesize a constructively Correct program directly from the specification?

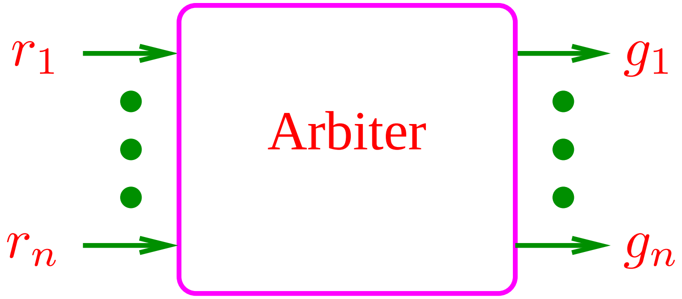
A Brief History of System Synthesis

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

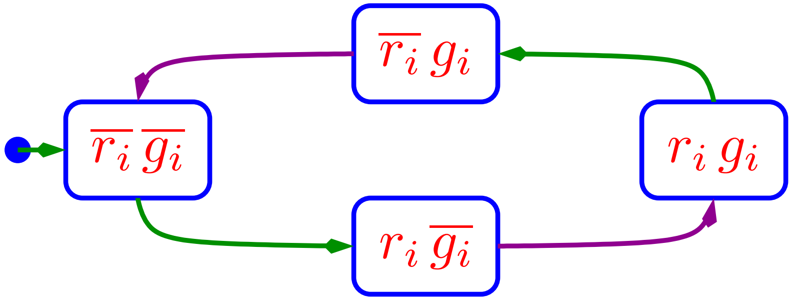
- Determine if there exists an automaton (sequential circuit) which realizes the specification.
- If the specification is realizable, construct an implementing circuit

The specification was given in the sequence calculus which is an explicit-time temporal logic.

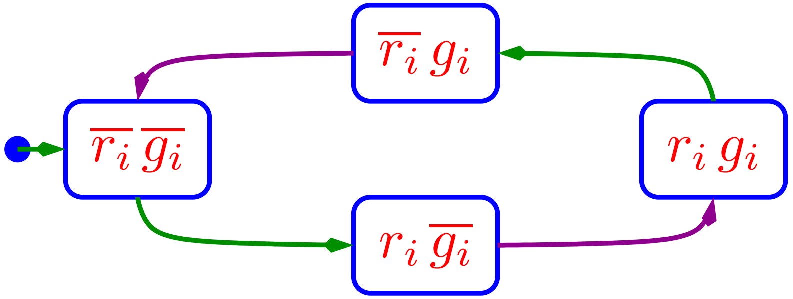
Example of a Specification: Arbiter



The protocol for each client:



The Behavioral Specification



$$\bigwedge_i \forall t : (r_i[t] = g_i[t] \rightarrow g_i[t+1] = g_i[t]) \;\wedge\; (r_i[t] \neq g_i[t] \rightarrow r_i[t+1] = r_i[t]) \quad \wedge$$
$$\bigwedge_{i \neq j} \forall t : \neg(g_i[t] \wedge g_j[t]) \quad \wedge$$
$$\bigwedge_i \forall t : r_i[t] \neq g_i[t] \rightarrow \exists s \geq t : r_i[s] = g_i[s]$$

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 $x = y^2$ 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 ω-automata were invented for this work.

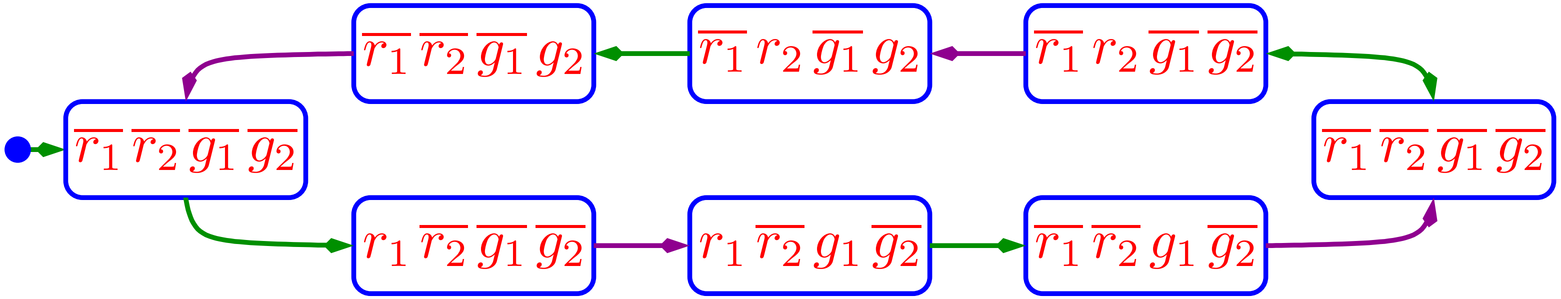
At the same year, Büchi and Landweber provided another solution, based on infinite games.

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 φ 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:


$\overline{r_1}\,\overline{r_2}\,\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,\overline{r_2}\,\overline{g_1}\,g_2$
$\overline{r_1}\,r_2\,\overline{g_1}\,g_2$
$\overline{r_1}\,r_2\,\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,\overline{r_2}\,\overline{g_1}\,\overline{g_2}$
$r_1\,\overline{r_2}\,\overline{g_1}\,\overline{g_2}$
$r_1\,\overline{r_2}\,g_1\,\overline{g_2}$
$\overline{r_1}\,\overline{r_2}\,g_1\,\overline{g_2}$

Such solutions are acceptable only in circumstances when the environment fully cooperate with the system.

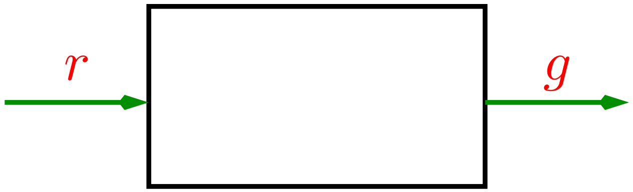
Next Step: Realizability $\sqsubset$ Satisfiability

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

Inconsistency

$$\Diamond\, g \quad \wedge \quad \Box\, \neg g$$

Unrealizability For a system



Realizing the specification

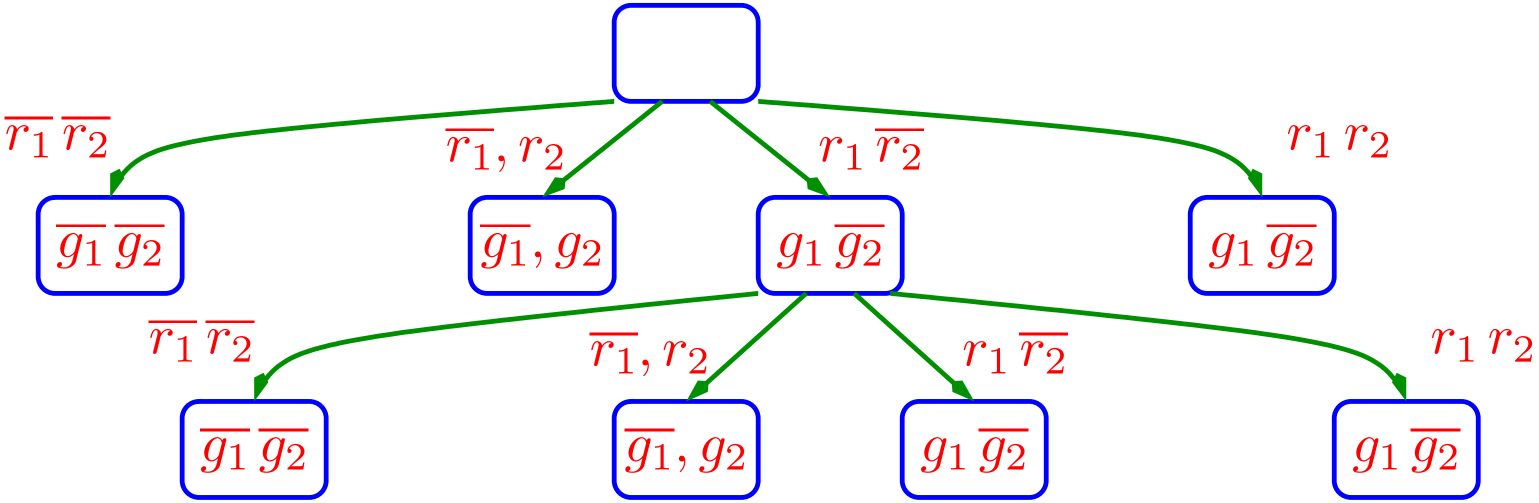
$$g \quad \longleftrightarrow \quad \Diamond\, r$$

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) environments.

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



Can be formulated as satisfaction of the CTL* formula

$$\mathbf{A}\varphi \;\wedge\; \mathbf{A}\square\;(\mathbf{EX}(\overline{r_1} \wedge \overline{r_2}) \;\wedge\; \mathbf{EX}(\overline{r_1} \wedge r_2) \;\wedge\; \mathbf{EX}(r_1 \wedge \overline{r_2}) \wedge \mathbf{EX}(r_1 \wedge r_2))$$

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 φ into a non-deterministic Büchiautomaton. The second exponent is due to the determinization of the automaton.

This result doomed synthesis to be considered highly untractable.

Simple Cases of Lower Complexity

In 1989, Ramadge and Wonham introduced the notion of controller synthesis and showed that for a specification of the form $\Box\, p$, 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 $\Box\, p$ 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 φ.
- 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 response.
- Temporal Testers and general LTL Model Checking.
- Controller Synthesis via Games.
- Synthesis from Recurrence Specifications.
- Synthesis from Reactivity Specifications. – The general case.

Fair Discrete Systems

As our computational model, we take fair discrete systems. An FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of:

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

- Θ – An initial condition. A satisfiable assertion that characterizes the initial states.

- ρ – 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 $x := x + 1$.

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

- $\mathcal{C} = \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle\}$ 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

$$x,\ y:\ \textbf{natural initially}\ x = y = 0$$

$$\begin{bmatrix} \ell_0: & \textbf{while}\ x = 0\ \textbf{do} \\ & [\ell_1:\ y := y+1] \\ \ell_2: & \end{bmatrix} \quad \| \quad \begin{bmatrix} m_0: & x := 1 \\ m_1: & \end{bmatrix}$$

$$-\quad P_1 \quad - \qquad\qquad\qquad -\quad P_2 \quad -$$

The Corresponding FDS

- State Variables $V: \begin{pmatrix} x, y & : & \text{natural} \\ \pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 & : & \{m_0, m_1\} \end{pmatrix}$.

- Initial condition: $\Theta : \ \pi_1 = \ell_0 \ \wedge \ \pi_2 = m_0 \ \wedge \ x = y = 0$.

- Transition Relation: ρ: $\rho_I \vee \rho_{\ell_0} \vee \rho_{\ell_1} \vee \rho_{m_0}$, with appropriate disjunct (transition) for each statement. For example, the disjuncts ρ_I and ρ_{ℓ_0} are

$$\rho_I : \quad \pi_1' = \pi_1 \ \wedge \ \pi_2' = \pi_2 \ \wedge \ x' = x \ \wedge \ y' = y$$

$$\rho_{\ell_0} : \quad \pi_1 = \ell_0 \ \wedge \ \begin{pmatrix} x = 0 \ \wedge \ \pi_1' = \ell_1 \\ \vee \\ x \neq 0 \ \wedge \ \pi_1' = \ell_2 \end{pmatrix} \ \wedge \ \pi_2' = \pi_2 \ \wedge \ x' = x \ \wedge \ y' = y$$

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

- Compassion set: $\mathcal{C}$: $\emptyset$.

Computations

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

$$\langle s, s' \rangle \models \rho_{\mathcal{D}}(V, V').$$

We define a computation of $\mathcal{D}$ to be an infinite sequence of states

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

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 $\mathcal{D}$-successor of the state s_j.
- Justice: For each $J \in \mathcal{J}$, σ 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 σ 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 $\mathcal{D}_P$ corresponding to a program P gives rise to a set of computations $\mathcal{C}omp(P) = \mathcal{C}omp(\mathcal{D}_P)$.

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

$$\begin{array}{l}\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\ ;\ x\colon 0\,,\ y\colon 0\rangle \xrightarrow{m_0} \langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_1\ ;\ x\colon 1\,,\ y\colon 0\rangle \xrightarrow{\ell_0} \\ \langle \pi_1\colon \ell_2\,,\ \pi_2\colon m_1\ ;\ x\colon 1\,,\ y\colon 0\rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots\end{array}$$

The following computation corresponds to the case that statement ℓ_1 is executed before m_0.

$$\begin{array}{l}\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\ ;\ x\colon 0\,,\ y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1\,,\ \pi_2\colon m_0\ ;\ x\colon 0\,,\ y\colon 0\rangle \xrightarrow{\ell_1} \\ \langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\ ;\ x\colon 0\,,\ y\colon 1\rangle \xrightarrow{m_0} \langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_1\ ;\ x\colon 1\,,\ y\colon 1\rangle \xrightarrow{\ell_0} \\ \langle \pi_1\colon \ell_2\,,\ \pi_2\colon m_1\ ;\ x\colon 1\,,\ y\colon 1\rangle \xrightarrow{\tau_I} \cdots \xrightarrow{\tau_I} \cdots\end{array}$$

In a similar way, we can construct for each $n \geq 0$ a computation that executes the body of statement ℓ_0 n times and then terminates in the final state

$$\langle \pi_1\colon \ell_2\,,\ \pi_2\colon m_1\ ;\ x\colon 1\,,\ y\colon n\rangle.$$

A Non-Computation

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

Thus, the sequence

$$
\begin{array}{l}
\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 0\rangle \xrightarrow{\ell_1} \\
\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 1\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 1\rangle \xrightarrow{\ell_1} \\
\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 2\rangle \xrightarrow{\ell_0} \langle \pi_1\colon \ell_1\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 2\rangle \xrightarrow{\ell_1} \\
\langle \pi_1\colon \ell_0\,,\ \pi_2\colon m_0\,;\ x\colon 0\,,\ y\colon 3\rangle \xrightarrow{\ell_0} \cdots
\end{array}
$$

in which statement m_0 is never executed is not an admissible computation. This is because it violates the justice requirement $\neg$*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 MUX-SEM, implements mutual exclusion by semaphores.

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

$$P_1 :: \left[\begin{array}{l} \ell_0:\ \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1: & \textbf{Non-critical} \\ \ell_2: & \textbf{request}\ y \\ \ell_3: & \textbf{Critical} \\ \ell_4: & \textbf{release}\ y \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0:\ \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} m_1: & \textbf{Non-critical} \\ m_2: & \textbf{request}\ y \\ m_3: & \textbf{Critical} \\ m_4: & \textbf{release}\ y \end{array}\right] \end{array}\right]$$

The compassion set of this program consists of

$$\mathcal{C}: \quad \{(\textit{at\_}\ell_2 \ \wedge\ y>0,\ \textit{at\_}\ell_3), \quad (\textit{at\_}m_2 \ \wedge\ y>0,\ \textit{at\_}m_3)\}.$$

Usually, with a compassionate transition τ, we associate the compassion requirement

$$(En(\tau), \quad taken(\tau))$$

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 ℓ_3 while P_2 is at m_3.
- Accessibility – Whenever process P_1 is at ℓ_2, it shall eventually reach it's critical section at ℓ_3. Similar requirement for P_2.

Consider the state sequence:

$$
\begin{array}{lllllll}
\sigma: & \langle \ell_0,\ m_0,\ 1\rangle & \longrightarrow & \cdots & \longrightarrow & \boxed{\langle \ell_2,\ m_2,\ 1\rangle} & \xrightarrow{m_2} \\
 & \langle \ell_2,\ m_3,\ 0\rangle & \xrightarrow{m_3} & \langle \ell_2,\ m_4,\ 0\rangle & \xrightarrow{m_4} & & \\
 & \langle \ell_2,\ m_0,\ 1\rangle & \xrightarrow{m_0} & \langle \ell_2,\ m_1,\ 1\rangle & \xrightarrow{m_1} & \boxed{\langle \ell_2,\ m_2,\ 1\rangle} & \xrightarrow{m_2} \\
 & \langle \ell_2,\ m_3,\ 0\rangle & \longrightarrow & \cdots\ , & & &
\end{array}
$$

which violates accessibility for process P_1. Due to the requirement of compassion for ℓ_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 $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel \mathcal{D}_2$, is given by $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lcl}
V & = & V_1 \quad \cup \quad V_2 \\
\Theta & = & \Theta_1 \quad \wedge \quad \Theta_2 \\
\rho & = & (\rho_1 \wedge \mathit{pres}(V_2 - V_1)) \;\vee\; (\rho_2 \wedge \mathit{pres}(V_1 - V_2)) \\
\mathcal{J} & = & \mathcal{J}_1 \quad \cup \quad \mathcal{J}_2 \\
\mathcal{C} & = & \mathcal{C}_1 \quad \cup \quad \mathcal{C}_2
\end{array}$$

The predicate $\mathit{pres}(U)$ 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. $\mathcal{D}(P_1 \parallel P_2) \quad \sim \quad \mathcal{D}(P_1) \parallel \mathcal{D}(P_2)$

Synchronous Parallel Composition

The synchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \mathrel{|||} \mathcal{D}_2$, is given by the FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lclcl}
V & = & V_1 & \cup & V_2 \\
\Theta & = & \Theta_1 & \wedge & \Theta_2 \\
\rho & = & \rho_1 & \wedge & \rho_2 \\
\mathcal{J} & = & \mathcal{J}_1 & \cup & \mathcal{J}_2 \\
\mathcal{C} & = & \mathcal{C}_1 & \cup & \mathcal{C}_2
\end{array}$$

Synchronous parallel composition can be used for hardware verification, where it is the natural operator for combining two circuits into a composed circuit. Here we use it for model checking of LTL formulas.

Claim 2. *The sequence σ of V-states is a computation of the combined $\mathcal{D}_1 \mathrel{|||} \mathcal{D}_2$ iff $\sigma\Downarrow_{V_1}$ is a computation of $\mathcal{D}_1$ and $\sigma\Downarrow_{V_2}$ is a computation of $\mathcal{D}_2$.*

Here, $\sigma\Downarrow_{V_i}$ denotes the sequence obtained from σ by restricting each of the states to a V_i-state.

Feasibility and Viability of Systems

An FDS $\mathcal{D}$ is said to be feasible if $\mathcal{D}$ has at least one computation.

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

The FDS $\mathcal{D}$ is defined to be viable if any finite run of $\mathcal{D}$ can be extended to a computation of $\mathcal{D}$.

Claim 3. *Every FDS derived from an SPL program is viable.*

Note that if $\mathcal{D}$ is a viable system, such that its initial condition $\Theta_{\mathcal{D}}$ is satisfiable, then $\mathcal{D}$ is feasible.

Requirement Specification Language: Temporal Logic

Assume an underlying (first-order) assertion language. The predicate at_ℓ_i, abbreviates the formula $\pi_j = \ell_i$, 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 $\vee$ and the basic temporal operators:

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

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

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

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

Semantics of LTL

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

- For an assertion p,
  $(\sigma, j) \models p \quad \Longleftrightarrow \quad s_j \models p$
  That is, we evaluate p locally on state s_j.
- $(\sigma, j) \models \neg p \quad \Longleftrightarrow \quad (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \quad \Longleftrightarrow \quad (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \quad \Longleftrightarrow \quad (\sigma, j+1) \models p$
- $(\sigma, j) \models p \, \mathcal{U} \, q \quad \Longleftrightarrow$ for some $k \geq j, (\sigma, k) \models q$,
  and for every i such that $j \leq i < k, \quad (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \quad \Longleftrightarrow \quad j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \, \mathcal{S} \, q \quad \Longleftrightarrow$ for some $k \leq j, (\sigma, k) \models q$,
  and for every i such that $j \geq i > k, \quad (\sigma, i) \models p$

This implies the following semantics for the derived operators:

- $(\sigma, j) \models \square\, p \quad \Longleftrightarrow \quad (\sigma, k) \models p$ for all $k \geq j$
- $(\sigma, j) \models \Diamond\, p \quad \Longleftrightarrow \quad (\sigma, k) \models p$ for some $k \geq j$

If $(\sigma, 0) \models p$ we say that p holds over σ and write $\sigma \models p$. Formula p is satisfiable if it holds over some model. Formula p is (temporally) valid if it holds over all models.

Formulas p and q are equivalent, denoted $p \sim q$, if $p \leftrightarrow q$ is valid. They are called congruent, denoted $p \approx q$, if $\square\ (p \leftrightarrow q)$ is valid. If $p \approx q$ then p can be replaced by q in any context.

The entailment $p \Rightarrow q$ is an abbreviation for $\square\ (p \ \rightarrow\ q)$.

For an FDS $\mathcal{D}$ and an LTL formula φ, we say that φ is $\mathcal{D}$-valid, denoted $\mathcal{D} \models \varphi$, if all computations of $\mathcal{D}$ satisfy φ.

Reading Exercises

Following are some temporal formulas φ and a verbal formulation of the constraint they impose on 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 $j \geq 0$.
- $\Box\,(p \;\rightarrow\; \Diamond\, q)$ — Every p is followed by a q. Can also be written as $p \Rightarrow \Diamond\, q$.
- $\Box\,\Diamond\, q$ — The sequence σ contains infinitely many q's.
- $\Diamond\,\Box\, q$ — All but finitely many states in σ satisfy q. Property q eventually stabilizes.
- $q \Rightarrow \diamondminus\, p$ — Every q is preceded by a p — causality.
- $(\neg r)\ \mathcal{W}\ q$ — q precedes r. r cannot occur before q — precedence. Note that q is not guaranteed, but r cannot happen without a preceding q.
- $(\neg r)\ \mathcal{W}\ (q \wedge \neg r)$ — q strongly precedes r.
- $p \Rightarrow (\neg r)\ \mathcal{W}\ q$ — Following every p, q precedes r.

Classification of Formulas/Properties

A formula of the form $\square\, p$ for some past formula p is called a safety formula.

A formula of the form $\square \diamond p$ for some past formula p is called a response formula.

An equivalent characterization is the form $p \Rightarrow \diamond q$. The equivalence is justified by

$$\square\,(p \rightarrow \diamond q) \qquad \sim \qquad \square \diamond ((\neg p)\ \mathcal{B}\ q)$$

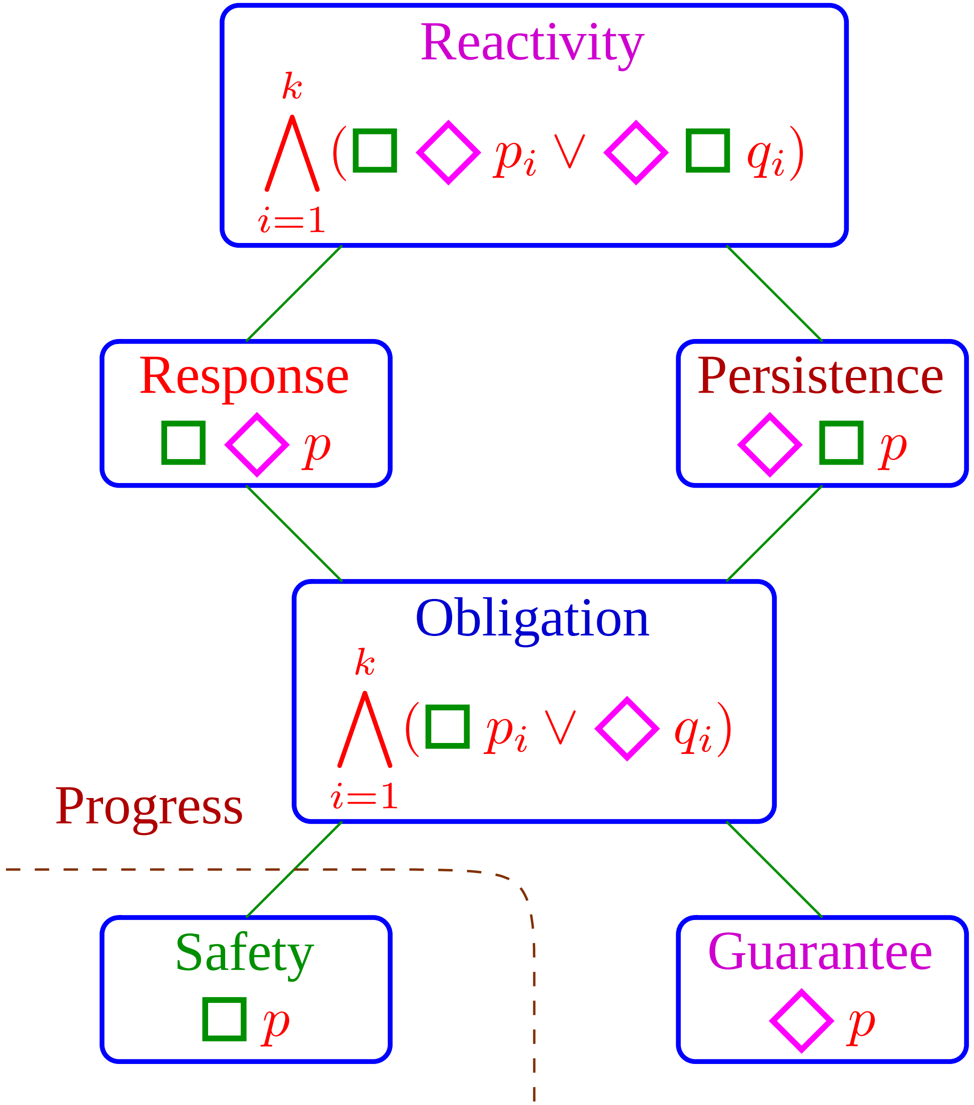
Both formulas state that either there are infinitely many q's, or there there are no p's, or there is a last q-position, beyond which there are no further p's.

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

Every temporal formula is equivalent to a conjunction of a reactivity formulas, i.e.

$$\bigwedge_{i=1}^{k} (\square \diamond p_i \ \vee\ \diamond \square\, q_i)$$

Hierarchy of the Temporal Properties



Reactivity

$$\bigwedge_{i=1}^{k}(\Box\Diamond p_i \vee \Diamond\Box q_i)$$

Response

$\Box\Diamond p$

Persistence

$\Diamond\Box p$

Obligation

$$\bigwedge_{i=1}^{k}(\Box p_i \vee \Diamond q_i)$$

Progress

Safety

$\Box p$

Guarantee

$\Diamond p$

where p, p_i, q, q_i are past formulas.

Temporal Specification of Properties

Formula φ is $\mathcal{D}$-valid, denoted $\mathcal{D} \models \varphi$, if all initial states of $\mathcal{D}$ satisfy φ. Such a formula specifies a property of $\mathcal{D}$.

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

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

$$P_1 :: \left[\begin{array}{l} \ell_0 : \ \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-critical} \\ \ell_2 : & \textbf{request}\ y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release}\ y \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \ \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} m_1 : & \textbf{Non-critical} \\ m_2 : & \textbf{request}\ y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release}\ y \end{array}\right] \end{array}\right]$$

- Mutual Exclusion – No computation of the program can include a state in which process P_1 is at ℓ_3 while P_2 is at m_3. Specifiable by the formula

$$\square\, \neg(at\_\ell_3 \ \wedge \ at\_m_3)$$

- Accessibility for P_1 – Whenever process P_1 is at ℓ_2, it shall eventually reach its critical section at ℓ_3. Specifiable by the formula

$$\square\, (at\_\ell_2 \quad \rightarrow \quad \diamondsuit\, at\_\ell_3)$$

Lectures Outline

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

Model Checking

This is a process by which we algorithmically check that a given finite state FDS D satisfies its temporal specification φ. 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 continuously work with assertions which characterize sets of states.

Here, we consider the symbolic approach. Note that every assertion over a finite-domain FDS can be represented as a boolean formula over boolean variables. Assume that a finite-state FDS is represented by such formulas, including the initial condition Θ and the bi-assertion ρ representing the transition relation.

We assume that we have an efficient representation of boolean assertions, and efficient algorithms for manipulation of such assertions, including all the boolean operations as well as existential and universal quantification. Note that, for a boolean variable b,

$$\exists b : \varphi(b) = \varphi(0) \ \vee \ \varphi(1) \qquad\qquad \forall b : \varphi(b) = \varphi(0) \ \wedge \ \varphi(1)$$

Also assume that we can efficiently check whether a given assertion is valid, i.e., equivalent to 1.

Successors and Their Transitive Closure

For an assertions $\varphi(V)$ and a bi-assertion $R(V, V')$, we define the existential successor predicate transformer:

$$\varphi \diamond R \quad = \quad \textit{unprime}(\exists V : \varphi(V) \wedge R(V, V'))$$

Obviously

$$\|\varphi \diamond R\| \quad = \quad \{s \mid s \text{ is an } R\text{-successor of a } \varphi\text{-state}\}$$

For example

$$\begin{array}{lcl} (x = 0) \diamond (x' = x + 1) & = & \textit{unprime}(\exists x : x = 0 \;\wedge\; x' = x + 1) \quad \sim \\ \textit{unprime}(x' = 1) & \sim & x = 1 \end{array}$$

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

$$\varphi \diamond R^* \quad =$$
$$\varphi \;\vee\; \varphi \diamond R \;\vee\; (\varphi \diamond R) \diamond R \;\vee\; ((\varphi \diamond R) \diamond R) \diamond R \;\vee\; \cdots$$

Predecessors and Their Transitive Closure

For an assertions $\varphi(V)$ and a bi-assertion $R(V,V')$, we define the existential predecessor predicate transformer:

$$R \diamond \psi \quad = \quad \exists V' : R(V,V') \wedge \psi(V')$$

Obviously

$$\|R \diamond \varphi\| \quad = \quad \{s \mid s \text{ is an } R\text{-predecessor of a } \varphi\text{-state}\}$$

For example

$$(x' = x+1) \diamond (x = 1) \quad = \quad \exists x' : x' = x+1 \ \wedge \ x' = 1 \quad \sim \quad x = 0$$

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

$$R^* \diamond \varphi \quad = $$
$$\varphi \ \vee \ R \diamond \varphi \ \vee \ R \diamond (R \diamond \varphi) \ \vee \ R \diamond (R \diamond (R \diamond \varphi)) \ \vee \ \cdots$$

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

$$\|y_1\| \subseteq \|y_2\| \quad \text{implies} \quad \|f(y_1)\| \subseteq \|f(y_2)\|$$

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 $\vee$ and $\wedge$, and the predecessor operator $\rho \diamond p$ is monotonic.

Consider a fix-point equation

$$y = f(y) \tag{1}$$

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 4. *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:

$$(p \wedge \rho_{\mathcal{D}})^* \diamond q \quad = \quad \mu y.(q \vee p \wedge \rho_{\mathcal{D}} \diamond y)$$

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

$$\begin{array}{lcllcl}
y_0 & = & 0 & & & \\
y_1 & = & q \vee 0 & & = & q \\
y_2 & = & q \vee p \wedge \rho_{\mathcal{D}} \diamond y_1 & & = & q \vee p \wedge \rho_{\mathcal{D}} \diamond q \\
y_3 & = & q \vee p \wedge \rho_{\mathcal{D}} \diamond y_2 & & = & q \vee p \wedge \rho_{\mathcal{D}} \diamond q \vee p \wedge \rho_{\mathcal{D}} \diamond (p \wedge \rho_{\mathcal{D}} \diamond q) \\
 & & & & \cdots &
\end{array}$$

Characterizing the set of all states which initiate a p-path leading to a q-state.

A Symbolic Algorithm for Model Checking Invariance

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

```
        $new$ : assertion
1.  $new := \neg p$
2.  Fix $(new)$ do
3.      $new := new \vee (\rho_{\mathcal{D}} \diamond new)$
4.  return $\Theta_{\mathcal{D}} \wedge new$
```

where

Fix (y) **do** $S \quad = \quad old := \neg y$; **While** $(y \neq old)$ **do** [$old := y$; 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 $\mathcal{D}$ satisfies *Inv* (p).

An equivalent formulation is

return $\Theta_{\mathcal{D}} \wedge \mu y : \neg p \vee \rho_{\mathcal{D}} \diamond y$

Equivalent Iterations

There are several equivalent ways to compute the set of all eventual predecessors of an assertion φ:

$$\rho^* \diamond \varphi \qquad \sim$$

$$\mu X.(\varphi \ \lor \ \rho \diamond X) \qquad \sim$$

$$X := 0; \quad \textbf{Fix}(X)\ [X := \varphi \ \lor \ \rho \diamond X] \quad \sim$$

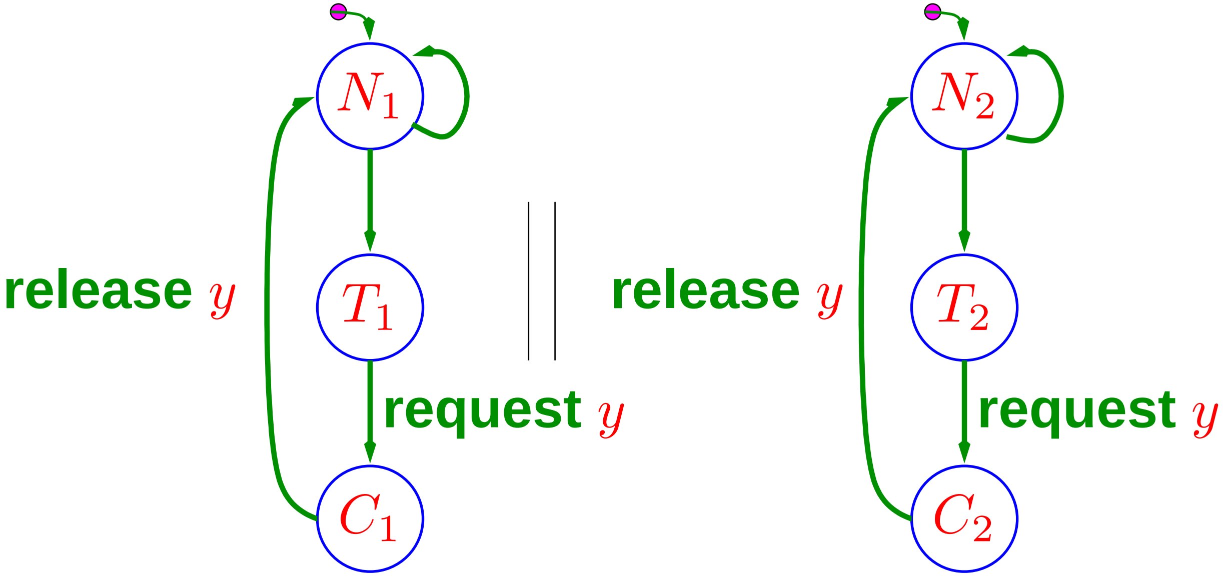
$$X := \varphi; \quad \textbf{Fix}(X)\ [X := X \ \lor \ \rho \diamond X] \quad \sim$$

$$\varphi \ \lor \ \rho \diamond \varphi \ \lor \ \rho \diamond (\rho \diamond \varphi) \ \lor \ \rho \diamond (\rho \diamond (\rho \diamond \varphi)) \ \lor \ \cdots$$

Example: a Simpler MUX-SEM

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

y: **natural initially** $y = 1$



The semaphore instructions **request** y and **release** y respectively stand for

$\langle$**when** $y = 1$ **do** $y := 0\rangle$ and $y := 1$.

Illustrate on MUX-SEM

We iterate as follows:

$$\begin{array}{ll}
\varphi_0: & \pi_1 = C \wedge \pi_2 = C \\
\varphi_1: & \varphi_0 \vee \left(\begin{array}{c} \cdots \\ \vee\, \pi_1 = T \wedge y = 1 \wedge \pi_1' = C \wedge y' = 0 \\ \vee\, \pi_2 = T \wedge y = 1 \wedge \pi_2' = C \wedge y' = 0 \end{array}\right) \diamond (\pi_1 = \pi_2 = C) \\
 & \sim \\
\end{array}$$

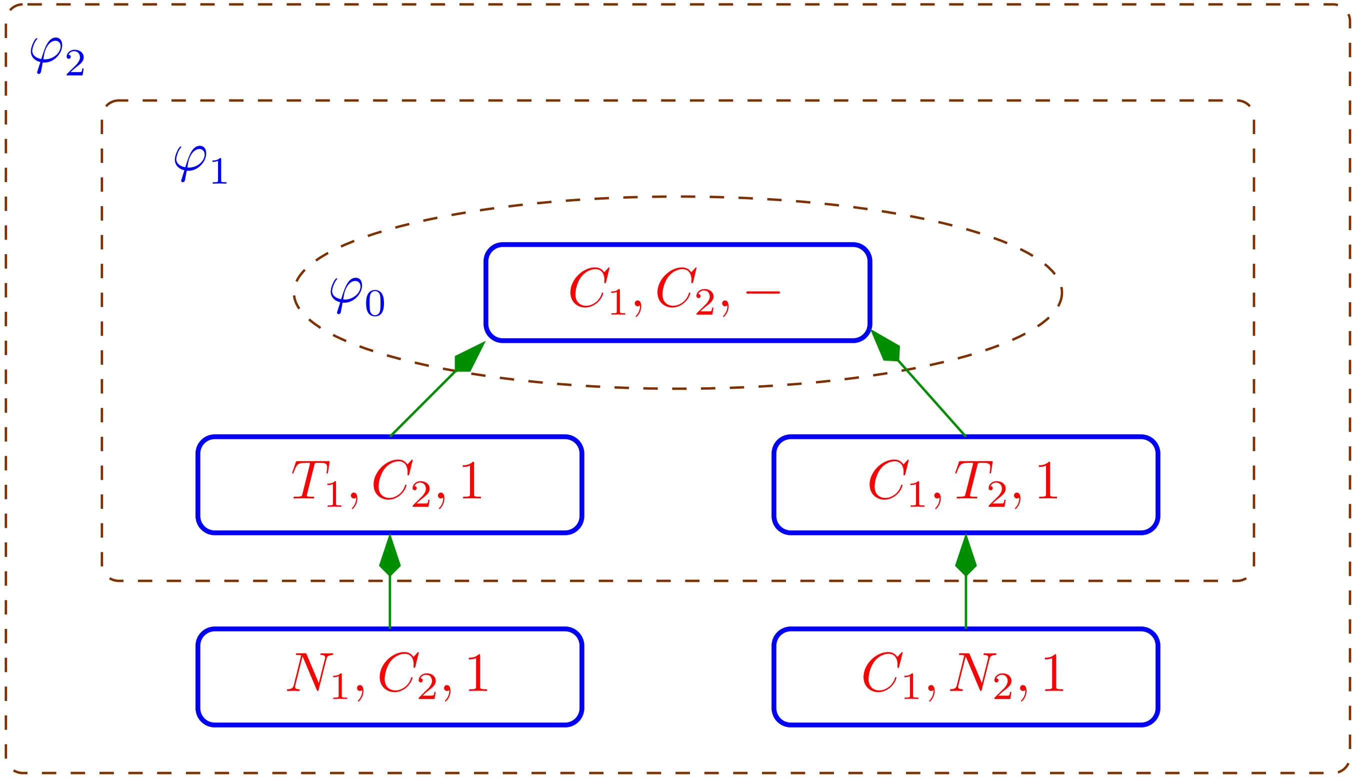
$$\pi_1 = \pi_2 = C \vee \pi_1 = T \wedge \pi_2 = C \wedge y = 1 \vee \pi_1 = C \wedge \pi_2 = T \wedge y = 1$$

$$\begin{array}{ll}
\varphi_2: & \varphi_1 \vee \pi_1 = N \wedge \pi_2 = C \wedge y = 1 \vee \pi_1 = C \wedge \pi_2 = N \wedge y = 1 \\
\varphi_3: & \varphi_2 \vee \pi_1 = C \wedge \pi_2 = C \wedge y = 0 \quad \sim \quad \varphi_2
\end{array}$$

The last equivalence is due to the general property $p \vee (p \wedge q) \sim p$.

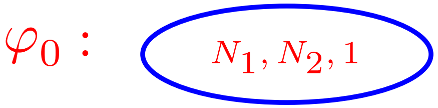
If we intersect φ_3 with the initial condition $\Theta: \pi_1 = N \wedge \pi_2 = N \wedge y = 1$ we obtain 0 (false). We conclude that MUX-SEM satisfies *Inv*$(\neg(\pi_1 = C \wedge \pi_2 = C))$.

Symbolic Exploration Progresses in Layers

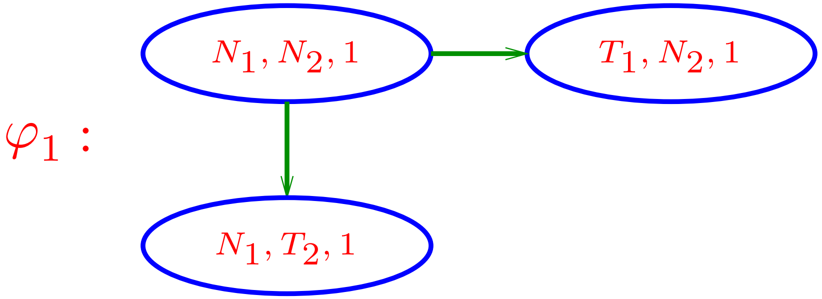


Illustrate Forward Exploration on MUX-SEM

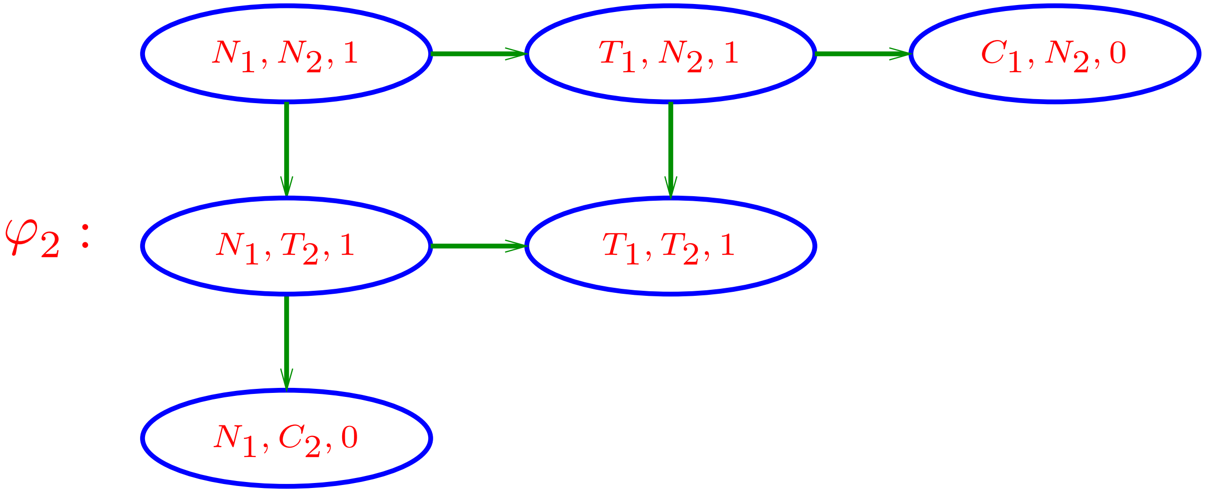
We iterate as follows:



Iteration 1:

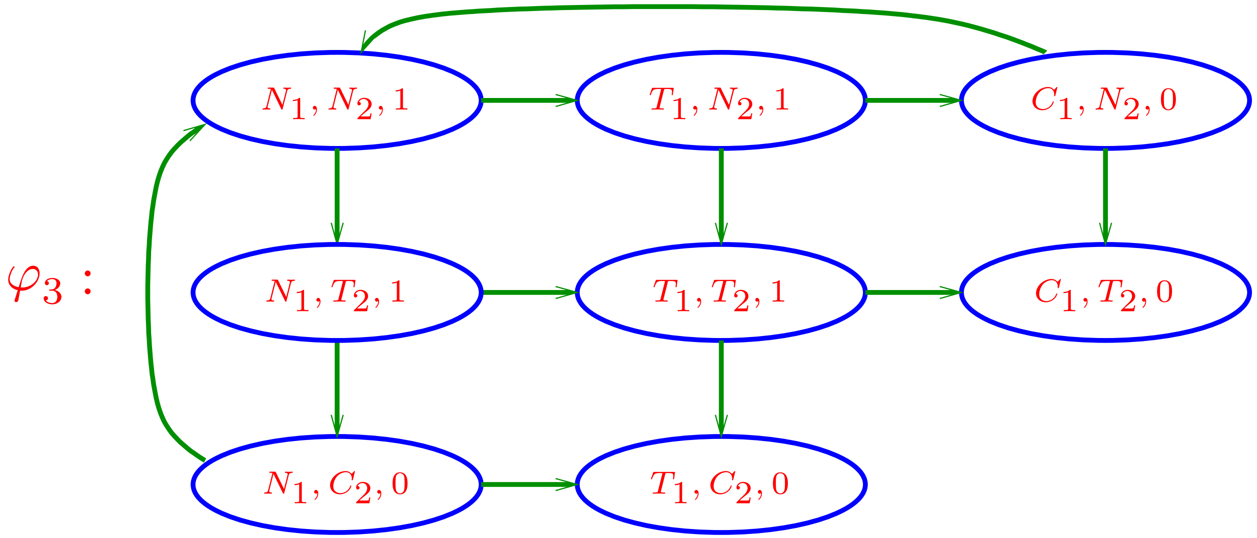


Iteration 2:

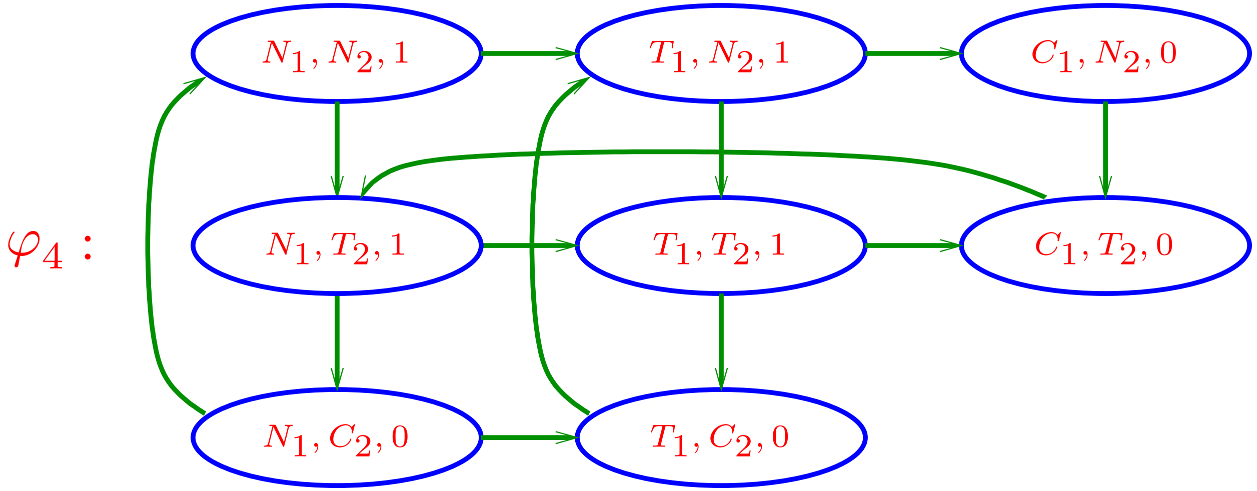


Forward Exploration Continued

Iteration 3:



Iteration 4 (Convergent):



Since last iteration does not intersect $C_1 \wedge C_2$, we conclude $\square \neg(C_1 \wedge C_2)$.

Checking for Feasibility

Before we discuss model checking response properties we discuss the problem of checking whether a given FDS is feasible.

A run of an FDS is an infinite sequence of states which satisfies the requirements of initiality and consecution but not necessarily any of the fairness requirements.

A state s of an FDS $\mathcal{D}$ is called reachable if it participates in some run of $\mathcal{D}$.

A state s is called feasible if it participates in some computation. The FDS is called feasible if it has at least one computation.

A set of states S is defined to be an F-set if it satisfies the following requirements:

F1. All states in S are reachable.

F2. Each state $s \in S$ has a ρ-successor in S.

F3. 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.

F4. For every state $s \in S$ and every compassion requirement $(p, q) \in \mathcal{C}$, either there exists an S-path leading from s to some q-state, or s satisfies $\neg p$.

F-Sets Imply Feasibility

Claim 5. [F-sets]

A reachable state s is feasible iff it has a path leading to some F-set.

Proof:

Assume that s is a feasible state. Then it participates in some computation σ. Let S be the (finite) set of all states that appear infinitely many times in σ. We will show that S is an F-set. It is not difficult to see that there exists a cutoff position $t \geq 0$ such that S contains all the states that appear at positions beyond t.

Obviously all states appearing in σ are reachable. If $s \in S$ appears in σ at position $i > t$ then it has a successor $s_{i+1} \in \sigma$ which is also a member of S.

Let $s = s_i \in \sigma$, $i > t$ be a member of S and $J \in \mathcal{J}$ be some justice requirement. Since σ is a computation it contains infinitely many J-positions. Let $k \geq i$ one of the J-positions appearing later than i. Then the path $s_i, \ldots, s_k$ is an S-path leading from s to a J-state.

Let $s = s_i \in \sigma$, $i > t$ be a member of S and $(p, q) \in \mathcal{C}$ be some compassion requirement. There are two possibilities by which σ may satisfy (p, q). Either σ contains only finitely many p-positions, or σ contains infinitely many q positions. It follows that either S contains no p-states, or it contains some q-states which appear infinitely many times in σ. In the first case, s satisfies $\neg p$. In the second case, there exists a path leading from s_i to s_k, a q-state such that $k \geq i$.

Proof Continued

In the other direction, assume the existence of an F-set S and a reachable state s which has a path leading to some state $s_1 \in S$. We will show that there exists a computation σ which contains s.

Since s is reachable and has a path leading to state $s_1 \in S$, there exists a finite sequence of states π leading from an initial state to s_1 and passing through s. We will show how π can be extended to a computation by an infinite repetition of the following steps. At any point in the construction, we denote by *end*(π) the state which currently appears last in π.

- We know that *end*$(\pi) \in S$ has a successor $s \in S$. Append s to the end of π.
- Consider in turn each of the justice requirements $J \in \mathcal{J}$. We append to π the S-path π_J connecting *end*(π) to a J-state.
- Consider in turn each of the compassion requirements $(p, q) \in \mathcal{C}$. If there exists an S-path π_q, connecting *end*(π) to a q-state, we append π_q to the end of π. Otherwise, we do not modify π. We observe that if there does not exist an S-path leading from *end*(π) to a q-state, then *end*(π) and all of its progeny within S must satisfy $\neg p$.

It is not difficult to see that the infinite sequence constructed in this way is a computation. ∎

Computing F-Sets

Assume an assertion φ which characterizes an F-set. Translating the requirements 1–4 into formulas, we obtain the following requirements:

$$\begin{array}{lcll}
\varphi & \rightarrow & \textit{reachable}_{\mathcal{D}} & \\
\varphi & \rightarrow & \rho \diamond \varphi & \text{Every } \varphi\text{-state has a } \varphi\text{-successor} \\
\varphi & \rightarrow & (\varphi \wedge \rho)^* \diamond (\varphi \wedge J) & \text{For every } J \in \mathcal{J} \\
\varphi & \rightarrow & \neg p \ \vee \ (\varphi \wedge \rho)^* \diamond (\varphi \wedge q) & \text{For every } (p,q) \in \mathcal{C}
\end{array}$$

This can be summarized as

$$\varphi \quad \rightarrow \quad \left(\begin{array}{lclcl}
\textit{reachable}_{\mathcal{D}} & \wedge & \rho \diamond \varphi & \wedge & \\
\bigwedge\limits_{J \in \mathcal{J}} (\varphi \wedge \rho)^* \diamond (\varphi \wedge J) & \wedge & \bigwedge\limits_{(p,q) \in \mathcal{C}} & \neg p \ \vee \ (\varphi \wedge \rho)^* \diamond (\varphi \wedge q) &
\end{array}\right)$$

Since we are interested in a maximal F-set, the computation can be expressed as:

$$\nu\varphi. \left(\begin{array}{lclcl}
\textit{reachable}_{\mathcal{D}} & \wedge & \rho \diamond \varphi & \wedge & \\
\bigwedge\limits_{J \in \mathcal{J}} (\varphi \wedge \rho)^* \diamond (\varphi \wedge J) & \wedge & \bigwedge\limits_{(p,q) \in \mathcal{C}} & \neg p \ \vee \ (\varphi \wedge \rho)^* \diamond (\varphi \wedge q) &
\end{array}\right)$$

Algorithmic Interpretation

Computing the maximal fix-point as a sequence of iterations, we can describe the computational process as follows:

Start by letting $\varphi := \textit{reachable}_{\mathcal{D}}$. Then repeat the following steps:

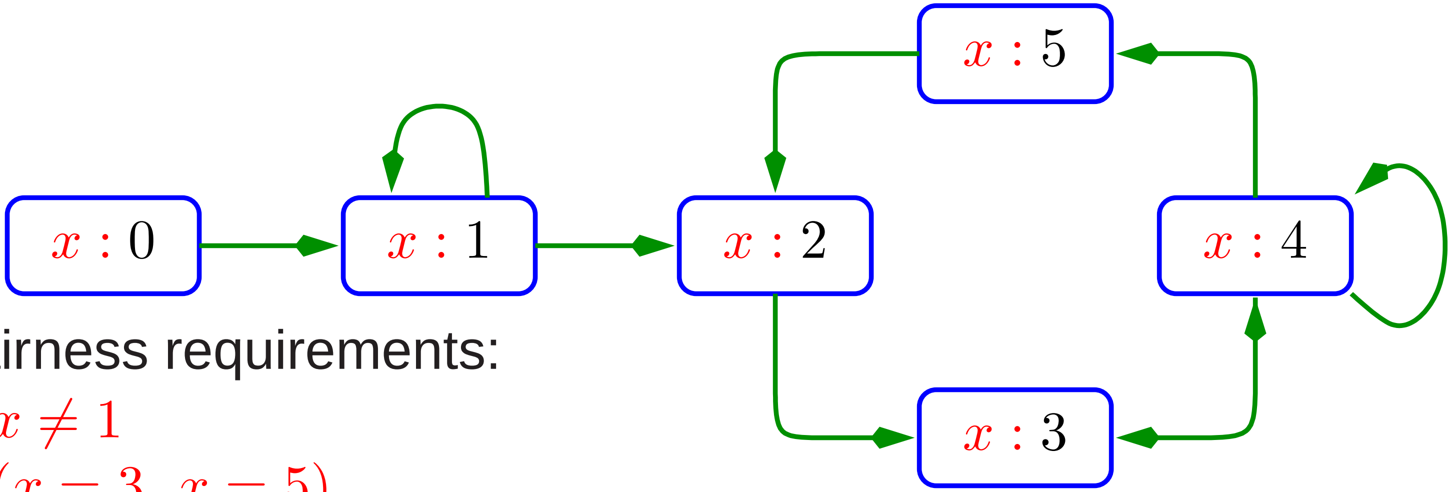
- Remove from φ all states which do not have a φ-successor.
- For each $J \in \mathcal{J}$, remove from φ all states which do not have a φ-path leading to a J-state.
- For each $(p, q) \in \mathcal{C}$, remove from φ all p-states which do not have a φ-path leading to a q-state.

until no further change.

To check whether an FDS $\mathcal{D}$ is feasible, we compute for it the maximal F-set and check whether it is empty. $\mathcal{D}$ is feasible iff the maximal F-set is not-empty.

Example

As an example, consider the following FDS:



with the fairness requirements:

$$\begin{array}{ll} J_1 : & x \neq 1 \\ C_1 : & (x = 3,\ x = 5) \\ C_2 : & (x = 2,\ x = 1) \end{array}$$

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

- Removing from φ_0 all $(x = 2)$-states which do not have a φ_0-path leading to an $(x = 1)$-state, we are left with $\varphi_1 : \{0, 1, 3, 4, 5\}$.
- Successively removing from φ_1 all states without successors, leaves $\varphi_2 : \{3, 4\}$.
- Removing from φ_2 all $(x = 3)$-states which do not have a φ_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.

We conclude that the above FDS is feasible.

Verifying Response Properties Through Feasibility Checking

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS and $p \Rightarrow \diamondsuit\, q$ be a response property we wish to verify over $\mathcal{D}$. Let *reachable*$_{\mathcal{D}}$ be the assertion characterizing all the reachable states in $\mathcal{D}$.

We define an auxiliary FDS $\mathcal{D}_{p,q} : \langle V, \Theta_{p,q}, \rho_{p,q}, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{ll} \Theta_{p,q} : & \textit{reachable}_{\mathcal{D}} \ \wedge\ p \ \wedge\ \neg q \\ \rho_{p,q} : & \rho \ \wedge\ \neg q' \end{array}$$

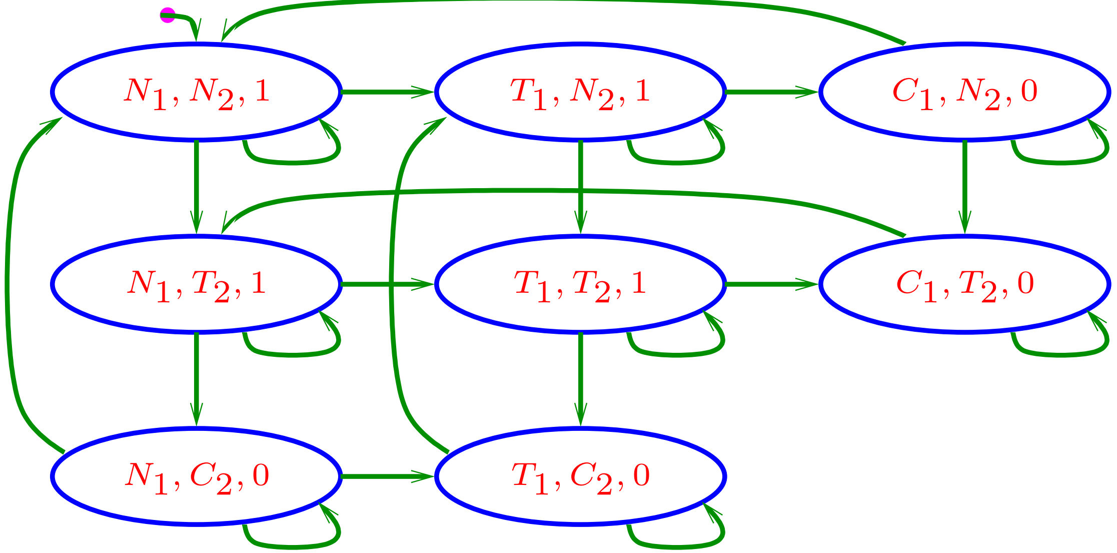
Thus, $\Theta_{p,q}$ characterizes all the $\mathcal{D}$-reachable p-states which do not satisfy q, while $\rho_{p,q}$ allows any ρ-step as long as the successor does not satisfy q.

Claim 6. [Model Checking Response]
$\mathcal{D} \models p \Rightarrow \diamondsuit\, q$ *iff* $\mathcal{D}_{p,q}$ *is unfeasible.*

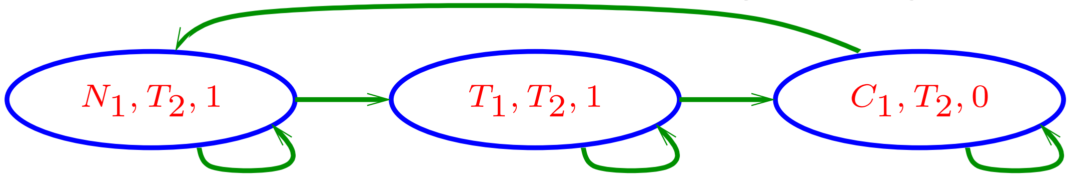
Proof: The claim is justifed by the observation that every computation of $\mathcal{D}_{p,q}$ can be extendable to a computation of $\mathcal{D}$ which violates the reponse property $p \Rightarrow \diamondsuit\, q$. Indeed, let $\sigma : s_k, s_{k+1}, \ldots$ be a computation of $\mathcal{D}_{p,q}$. By the definition of $\Theta_{p,q}$, we know that s_k is a $\mathcal{D}$-reachable p-state. Thus, there exists, a finite sequence $s_0, \ldots, s_k$, such that s_0 is $\mathcal{D}$-initial. The infinite sequence $s_0, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots$ is a computation of $\mathcal{D}$ which contains a p-state at position k, and has no following q-state. This sequence violates $p \Rightarrow \diamondsuit\, q$. ▟

Example: MUX-SEM

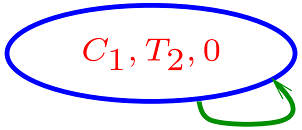
Following is the set of all reachable states of program MUX-SEM.



Assume we wish to verify the property $T_2 \Rightarrow \Diamond C_2$. We start by forming MUX-SEM$_{T_2,C_2}$, whose set of reachable states is given by:



First, we eliminate all $(T_2 \wedge y = 1)$-states which do not have a path leading to a C_2-state. This leaves us with:



Next, we eliminate all states which do not have a path leading to a $\neg C_1$-state. This leaves us with nothing. We conclude that MUX-SEM $\models T_2 \Rightarrow \Diamond C_2$.

Lectures Outline

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

Model Checking General Temporal Formulas

Next, we consider methods for model checking general LTL formulas.

Let $\mathcal{D}$ be an FDS and φ an LTL formula. Assume we wish to check whether $\mathcal{D} \models \varphi$. We proceed along the following steps:

- Construct a temporal acceptor $\mathcal{A}(\neg\varphi)$. This is an FDS whose computations are all the sequences falsifying φ.
- Form the parallel composition $\mathcal{D} \mid\mid\mid \mathcal{A}(\neg\varphi)$. This is an FDS whose computations are all computations of $\mathcal{D}$ which violate φ.
- Check whether the composition $\mathcal{D} \mid\mid\mid \mathcal{A}(\neg\varphi)$ is feasible. $\mathcal{D} \models \varphi$ iff $\mathcal{D} \mid\mid\mid T(\neg\varphi)$ is infeasible.

It only remains to describe the construction of an acceptor $\mathcal{A}(\psi)$ for a general LTL formula ψ.

Temporal Testers

The building blocks from which we construct acceptors are temporal testers. Let φ be a temporal formula over vocabulary U, and let $x \notin U$ be a boolean variable disjoint from U.

In the following, let $\sigma : s_0, s_1, \ldots$ be an infinite sequence of states over $U \cup \{x\}$. We say that x matches φ in σ if, for every position $j \geq 0$, the value of x at position j is true iff $(\sigma, j) \models \varphi$.

A temporal tester for φ is an FDS $T(\varphi)$ over $U \cup \{x\}$, satisfying the requirement:

The infinite sequence σ is a computation of $T(\varphi)$ iff x matches φ in σ.

A consequence of this definition is that every infinite sequence π of U-states can be extended into a computation σ of $T(\varphi)$ by interpreting x at position $j \geq 0$ of σ as 1 iff $(\pi, j) \models \varphi$.

We can view $T(\varphi)$ as a (possibly non-deterministic) transducer which incrementally reads the values of the variables U and outputs in x the current value of φ over the infinite sequence.

Construction of Temporal Testers

A formula φ is called a principally temporal formula (PTF) if the main operator of p is temporal. A PTF is called a basic temporal formula if it contains no other PTF as a proper sub-formula.

We start our construction by presenting temporal testers for the basic temporal formulas.

A Tester for $\bigcirc p$

The tester for the formula $\bigcirc p$ is given by:

$$T(\bigcirc p): \left\{ \begin{array}{rlll} V: & \mathit{Vars}(p) \cup \{x\} & & \\ \Theta: & 1 & & \\ \rho: & x & = & p' \\ \mathcal{J} = \mathcal{C}: & \emptyset & & \end{array} \right.$$

Claim 7.
$T(\bigcirc p)$ *is a temporal tester for* $\bigcirc p$*.*

Proof:
Let σ be a computation of $T(\bigcirc p)$. We will show that x matches $\bigcirc p$ in σ. Let $j \geq 0$ be any position. By the transition relation, $x = 1$ at position j iff $s_{j+1} \models p$ iff $(\sigma, j) \models \bigcirc p$.

Let σ be an infinite sequence such that x matches $\bigcirc p$ in σ. We will show that σ is a computation of $T(\bigcirc p)$. For any position $j \geq 0$, $x = 1$ at j iff $(\sigma, j) \models \bigcirc p$, iff $s_{j+1} \models p$. Thus, x satisfies $x = p'$ at every position j. ▟

A Tester for $p\mathcal{U}q$

The tester for the formula $p\mathcal{U}q$ is given by:

$$T(p\mathcal{U}q): \left\{ \begin{array}{rl} V: & \textit{Vars}(p,q) \ \cup \ \{x\} \\ \Theta: & 1 \\ \rho: & x \ = \ q \ \vee \ (p \ \wedge \ x') \\ \mathcal{J}: & q \ \vee \ \neg x \\ \mathcal{C}: & \emptyset \end{array} \right.$$

Claim 8. $T(p\mathcal{U}q)$ *is a temporal tester for* $p\mathcal{U}q$.

Proof:

Based on the expansion formula

$$p\mathcal{U}q \quad = \quad q \ \vee \ (p \ \wedge \ \bigcirc(p\mathcal{U}q))$$

Why Do We Need the Justice Requirement

Reconsider the temporal tester for $p\mathcal{U}q$:

$$T(p\mathcal{U}q): \left\{ \begin{array}{rl} V: & \textit{Vars}(p,q) \ \cup \ \{x\} \\ \Theta: & 1 \\ \rho: & x \quad = \quad q \ \vee \ (p \ \wedge \ x') \\ \mathcal{J}: & q \ \vee \ \neg x \\ \mathcal{C}: & \emptyset \end{array} \right.$$

We wish to show that the justice requirement $q \vee \neg x$ is essential for the correctness of the construction. Consider a state sequence $\sigma : s_0, s_1, \ldots$ in which q is identically false and p is identically true at all positions. Obviously, $(\sigma, j) \not\models p\,\mathcal{U}\,q$, for all $j \geq 0$, and the transition relation reduces to the equation

$$x = x'.$$

This equation has two possible solutions, one in which x is identically false and the other in which x is identically true at all positions. Only $x = 0$ matches $p\,\mathcal{U}\,q$. This is also the only solution which satisfies the justice requirement.

Thus, the role of the justice requirement is to select among several solutions to the transition relation equation, a unique one which matches the basic temporal formula at all positions.

A Tester for $p\mathcal{W}q$

A supporting evidence for the significance of the justice requirements is provided by the tester for the formula $p\mathcal{W}q$:

$$T(p\mathcal{W}q) : \begin{cases} V : & \textit{Vars}(p,q) \ \cup \ \{x\} \\ \Theta : & 1 \\ \rho : & x \ = \ q \ \vee \ (p \ \wedge \ x') \\ \mathcal{J} : & \neg p \ \vee \ x \\ \mathcal{C} : & \emptyset \end{cases}$$

Note that the transition relation of $T(p\mathcal{W}q)$ is identical to that of $T(p\mathcal{U}q)$, and they only differ in their respective justice requirements.

The role of the justice requirement in $T(p\mathcal{W}q)$ is to eliminate the solution $x = 0$ over a computation in which $p = 1$ and $q = 0$ at all positions.

Testers for the Derived Operators

Based on the testers for $\mathcal{U}$ and $\mathcal{W}$, we can construct testers for the derived operators $\diamondsuit$ and $\square$. They are given by

$$T(\diamondsuit\, p) : \left\{ \begin{array}{rl} V : & \textit{Vars}(p) \ \cup \ \{x\} \\ \Theta : & 1 \\ \rho : & x \ = \ p \ \vee \ x' \\ \mathcal{J} : & p \ \vee \ \neg x \\ \mathcal{C} : & \emptyset \end{array} \right. \qquad T(\square\, p) : \left\{ \begin{array}{rl} V : & \textit{Vars}(p) \ \cup \ \{x\} \\ \Theta : & 1 \\ \rho : & x \ = \ p \ \wedge \ x' \\ \mathcal{J} : & \neg p \ \vee \ x \\ \mathcal{C} : & \emptyset \end{array} \right.$$

A formula such as $\diamondsuit\, p$ can be viewed as a “promise for an eventual p”. The justice requirement $p \vee \neg x$ can be interpreted as suggesting:

Either fulfill all your promises or stop promising.

Note that once $x = 0$ in the tester $T(\diamondsuit\, p)$, it remains 0 and requires $p = 0$ ever after.

Testers for the Basic Past Formulas

The following are testers for the basic past formulas $\ominus p$ and $p\mathcal{S}q$:

$$T(\ominus p) : \left\{ \begin{array}{rl} V: & \mathit{Vars}(p) \;\cup\; \{x\} \\ \Theta: & x = 0 \\ \rho: & x' \quad = \quad p \\ \mathcal{J}: & \emptyset \\ \mathcal{C}: & \emptyset \end{array} \right. \qquad T(p\mathcal{S}q) : \left\{ \begin{array}{rl} V: & \mathit{Vars}(p,q) \;\cup\; \{x\} \\ \Theta: & x = q \\ \rho: & x' \quad = \quad q' \;\vee\; (p' \wedge x) \\ \mathcal{J}: & \emptyset \\ \mathcal{C}: & \emptyset \end{array} \right.$$

Note that testers for past formulas are not associated with any fairness requirements. On the other hand, they have a non-trivial initial conditions.

Testers for Compound Temporal Formulas

Up to now we only considered testers for basic formulas. The construction for non-basic formulas is based on the following reduction principle. Let $f(\varphi)$ be a temporal formula containing one or more occurrences of the basic formula φ. Then the temporal tester for $f(\varphi)$ can be constructed according to the following recipe:

$$T(f(\varphi)) \quad = \quad T(f(x_\varphi)) \parallel\!| \; T(\varphi)$$

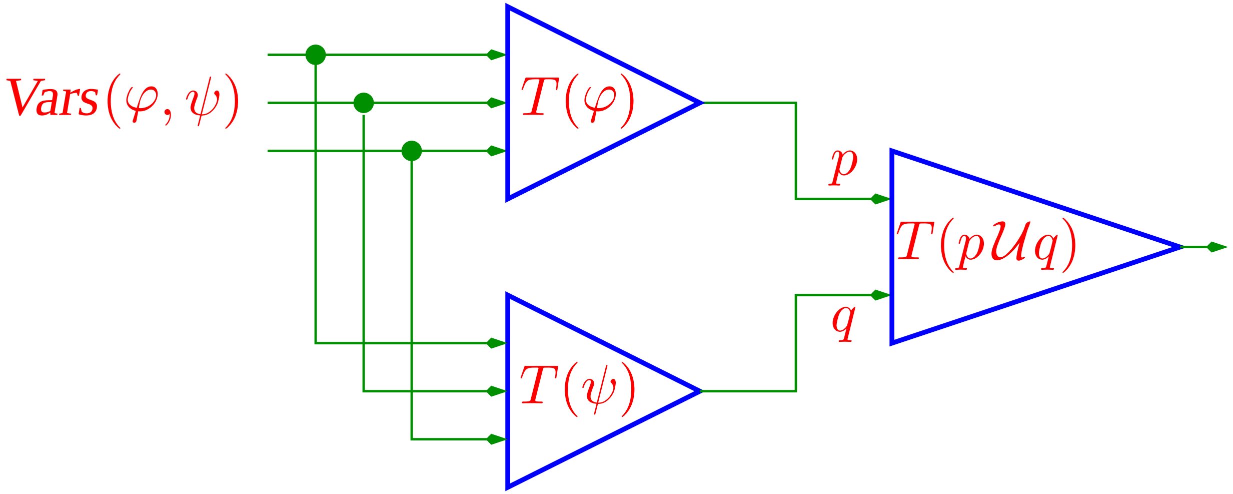
where, x_φ is the boolean output variable of $T(\varphi)$, and $f(x_\varphi)$ is obtained from $f(\varphi)$ by replacing every instance of φ by x_φ.

Following this recipe the temporal tester for an arbitrary formula f can be decomposed into a synchronous parallel composition of smaller testers, one for each basic formula nested within f.

When all possible substitution/composition steps are performed, we are left with a (non-temporal) assertion. We refer to this assertion as the redux of the original formula f, and denote it by redux(f).

Testers as Circuits

Having viewed testers as transducers, we can view their composition as a circuit interconnection. For example, in the following diagram we show how a tester for the compound formula $\varphi\mathcal{U}\psi$ can be constructed by interconnecting the testers for φ, ψ, and the tester for the basic formula $p\mathcal{U}q$.



Acceptors

Testers are the essential building blocks for the construction of an acceptor. An acceptor for an LTL formula φ (over variables U) is an FDS $\mathcal{A}(\varphi)$ such that

> The U-sequence σ satisfies φ iff σ is a U-projection of a computation of $\mathcal{A}(\varphi)$.

Thus, unlike testers, an acceptor only accepts at position 0.

The construction of an acceptor is defined recursively as follows:

- For an assertion p,

$$\mathcal{A}(p): \quad \left\{\begin{array}{ll} V: & \textit{Vars}(p) \\ \Theta: & p \\ \rho: & 1 \\ \mathcal{J} = \mathcal{C}: & \emptyset \end{array}\right.$$

- For a formula $f(\varphi)$ containing one or more occurrences of the basic formula φ,

$$\mathcal{A}(f(\varphi)) \quad = \quad \mathcal{A}(f(x_\varphi)) \;|||\; T(\varphi)$$

Example: An Acceptor for $\neg\Diamond\Box\, p$

Following is a tester for the formula $\neg\Diamond\Box\, p$ which is obtained by computing the parallel composition $\mathcal{A}(\neg x_{\Diamond})\ |||\ T(\Diamond\, x_{\Box})\ |||\ T(\Box\, p)$.

$$\mathcal{A}(\Diamond\Box\, p) : \left\{ \begin{array}{rl} V : & \textit{Vars}(p)\ \cup\ \{x_{\Diamond}, x_{\Box}\} \\ \Theta : & \neg x_{\Diamond} \\ \rho : & (x_{\Box} = p\ \wedge\ x_{\Box}')\ \wedge\ (x_{\Diamond} = x_{\Box}\ \vee\ x_{\Diamond}') \\ \mathcal{J} : & \{\neg p\ \vee\ x_{\Box},\quad x_{\Box}\ \vee\ \neg x_{\Diamond}\} \\ \mathcal{C} : & \emptyset \end{array} \right.$$

Note that the redux of $\neg\Diamond\Box\, p$ is $\neg x_{\Diamond}$.

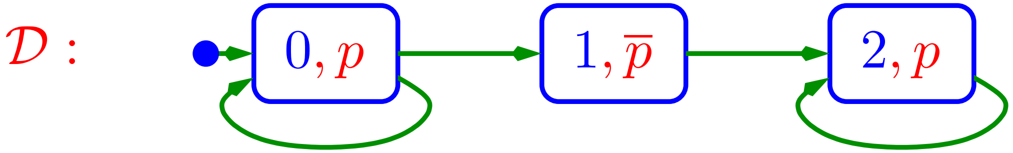
Model Checking General Temporal Formulas

To check whether $\mathcal{D} \models \varphi$, perform the following steps:

- Construct the acceptor $\mathcal{A}(\neg\varphi)$.
- Form the combined system $C = \mathcal{D} \parallel\!\mid \mathcal{A}(\neg\varphi)$.
- Check whether C is feasible.
- Conclude $\mathcal{D} \models \varphi$ iff C is infeasible.

Example

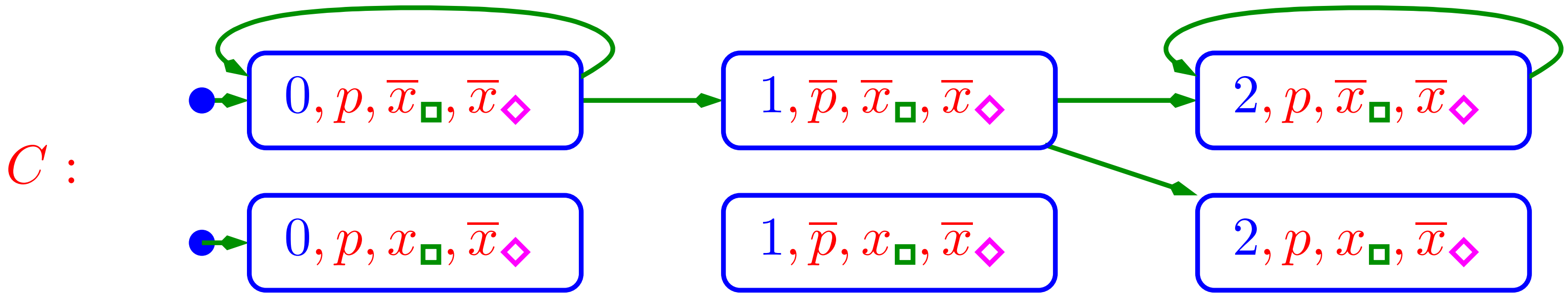
Consider the following system:



For which we wish to verify the property $\Diamond\Box\, p$.

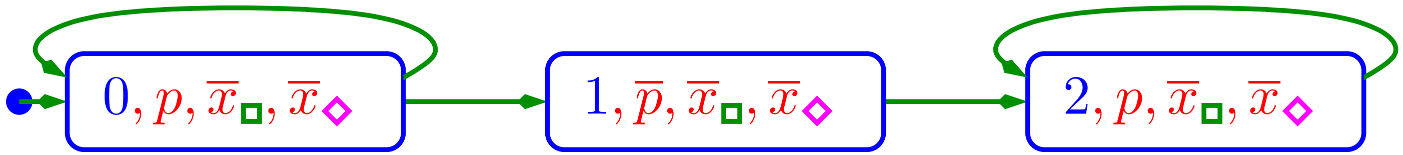
Example: Continued

Composing the system with the acceptor $\mathcal{A}(\neg\Diamond\Box\, p)$, we obtain:


$C:$
$0, p, \overline{x}_{\Box}, \overline{x}_{\Diamond}$
$1, \overline{p}, \overline{x}_{\Box}, \overline{x}_{\Diamond}$
$2, p, \overline{x}_{\Box}, \overline{x}_{\Diamond}$
$0, p, x_{\Box}, \overline{x}_{\Diamond}$
$1, \overline{p}, x_{\Box}, \overline{x}_{\Diamond}$
$2, p, x_{\Box}, \overline{x}_{\Diamond}$

with the justice requirements $\neg p \vee x_{\Box}$ and $x_{\Box} \vee \neg x_{\Diamond}$.

Eliminating all unreachable states and states with no successors, we are left with:


$0, p, \overline{x}_{\Box}, \overline{x}_{\Diamond}$
$1, \overline{p}, \overline{x}_{\Box}, \overline{x}_{\Diamond}$
$2, p, \overline{x}_{\Box}, \overline{x}_{\Diamond}$

State 2 is eliminated because it does not have a path leading to a $\neg p \vee x_{\Box}$ -state. Then state 1 is eliminated. having no successors. Finally, 0 is eliminated because it cannot reach a $\neg p \vee x_{\Box}$ -state. Nothing is left, hence the system satisfies the property $\Diamond\Box\, p$.

Correctness of the Algorithms

Claim 9.
For an FDS $\mathcal{D}$ *and temporal formula* φ, $\mathcal{D} \models \varphi$ *iff* $C : \mathcal{D} \parallel\!\mid \mathcal{A}(\neg\varphi)$ *is infeasible*

Proof:
The proof is based on the observation that every computation of the combined system C is a computation of $\mathcal{D}$ which satisfies the negation of φ. Therefore, the existence of such a computation shows that not all computations of $\mathcal{D}$ satisfy φ, and therefore, φ is not valid over $\mathcal{D}$. ▟