

Program Synthesis in Action

Amir Pnueli

New York University and Weizmann Institute of Sciences

Perspectives in Verification, November 18, 2005

Joint work with

Yonit Kesten, Nir Piterman, Yaniv Sa'ar,

Research Supported in part by SRC grant 2004-TJ-1256 and the European Union project Prosyd.

Motivation

Why **verify**, if we can automatically synthesize a program which is **correct by construction**?

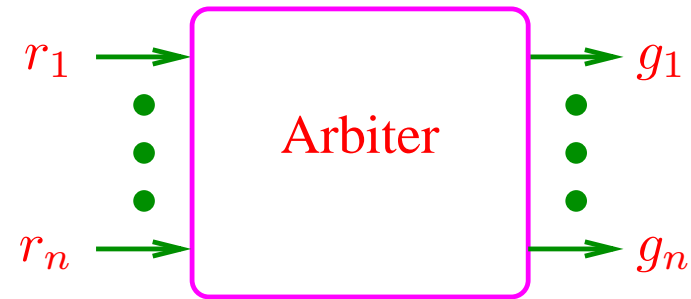
A Brief History of System Synthesis

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

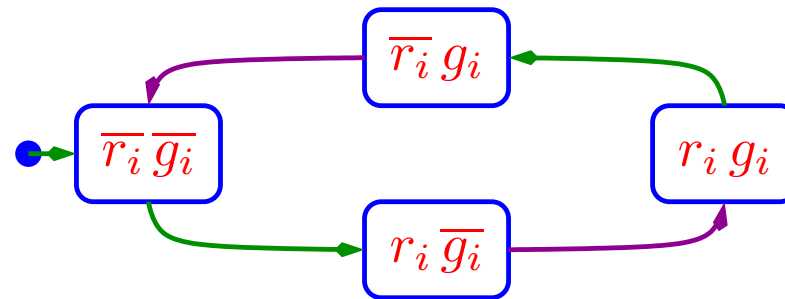
- Determine if there exists an automaton (sequential circuit) which realizes the specification.
- If the specification is realizable, construct an implementing circuit

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

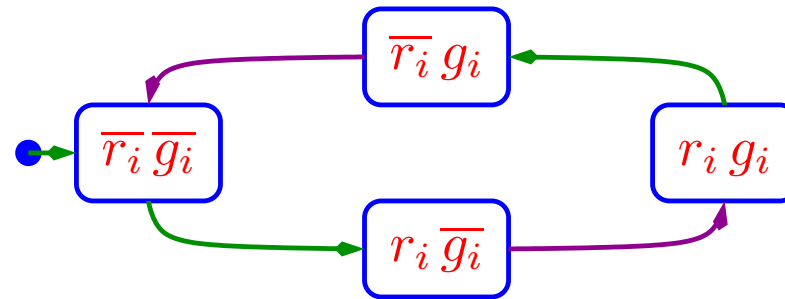
Example of a Specification: Arbiter



The protocol for each client:



The Behavioral Specification



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

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.

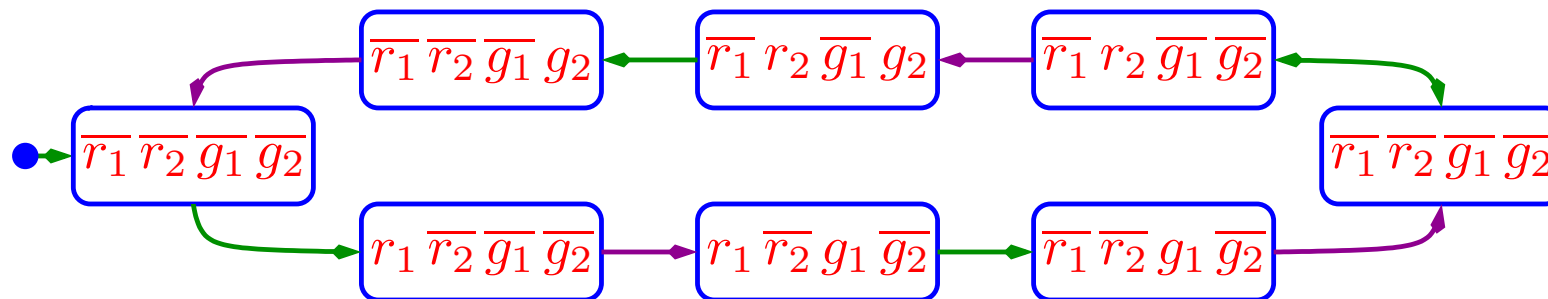
These two techniques (**Trees** and **Games**) are still the main techniques for performing synthesis.

Wolfgang Thomas' work contributed greatly to see the confluence of these two approaches.

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:



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

Next Step: Realizability \square Satisfiability

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

Inconsistency

$$\diamond g \wedge \square \neg g$$

Unrealizability For a system



Realizing the specification

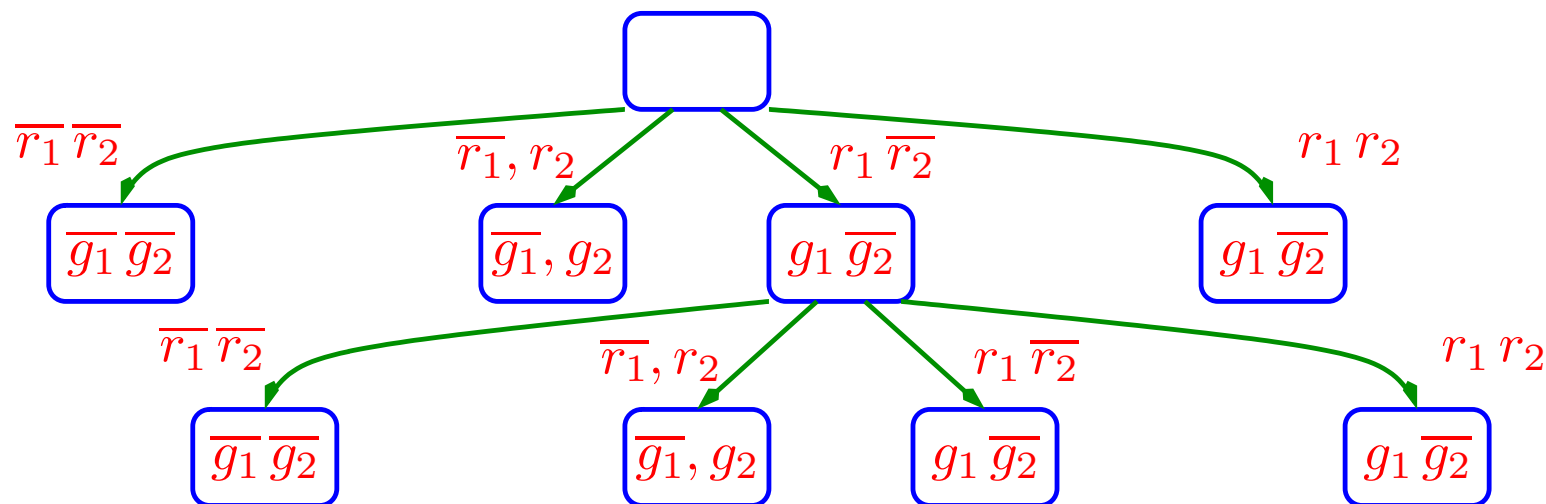
$$g \longleftrightarrow \diamond r$$

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 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 Büchi automaton. 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 $\square p$, 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.

Property-Based System Design

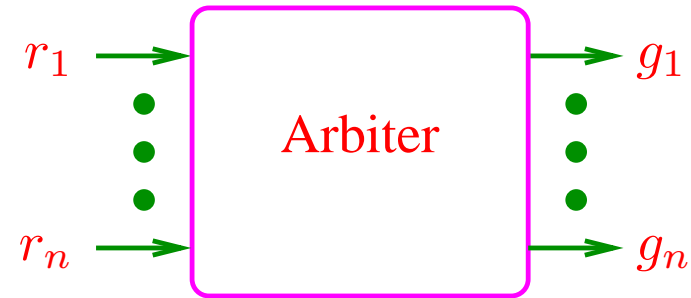
While the rest of the world seems to be moving in the direction of **model-based** design (see **UML**), some of us persisted with the vision of **property-based** approach.

Specification is stated declaratively as a set of **properties**, from which a **design** can be extracted.

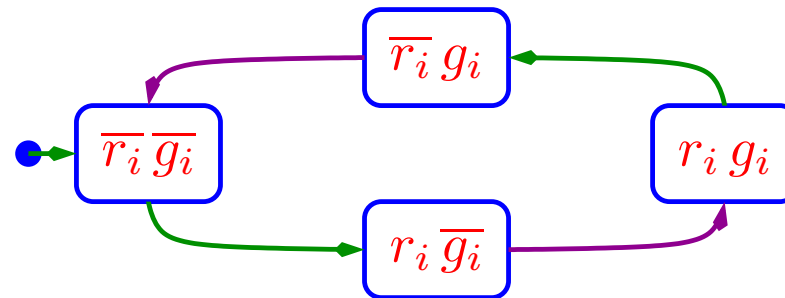
This is currently studied in the hardware-oriented European project **PROSYD**.

Example Specification

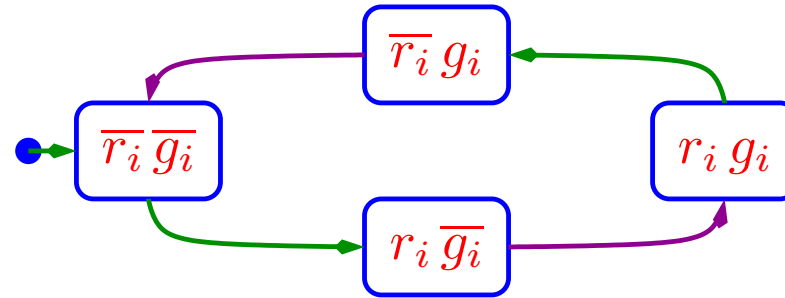
Reconsider a specification for an **arbiter**.



The protocol for each client:



The Specification



Assumptions (Constraints on the Environment)

$$A : \bigwedge_i (\overline{r_i} \wedge (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \wedge r_i \wedge g_i \Rightarrow \diamond \overline{r_i})$$

Guarantees (Expectations from System)

$$G : \bigwedge_{i \neq j} \square \neg (g_i \wedge g_j) \wedge \bigwedge_i \left(\overline{g_i} \wedge \left(\begin{array}{l} r_i = g_i \Rightarrow \bigcirc g_i = g_i \wedge \\ r_i \wedge \overline{g_i} \Rightarrow \diamond g_i \wedge \\ \overline{r_i} \wedge g_i \Rightarrow \diamond \overline{g_i} \end{array} \right) \right)$$

Total Specification

$$\varphi : A \rightarrow G$$

Program Synthesis Via Game Playing

A **game** is given by $\mathcal{G} : \langle V = X \cup Y, \Theta, \rho_1, \rho_2, \varphi \rangle$, where

- $V = X \cup Y$ are the **state variables**, with X being the **environment's** (player 1) variables, and Y being the **system's** (player 2) variables. A state of the game is an interpretation of V . Let Σ denote the set of all states.
- Θ — the **initial condition**. An assertion characterizing the initial states.
- $\rho_1(X, Y, X')$ — **Transition relation** for player 1 (**Environment**).
- $\rho_2(X, Y, X', Y')$ — **Transition relation** for player 2 (**system**).
- φ — The **winning condition**. An **LTL** formula characterizing the plays which are winning **for player 2**.

A state s_2 is said to be a **\mathcal{G} -successor** of state s_1 , if both $\rho_1(s_1[V], s_2[X])$ and $\rho_2(s_1[V], s_2[V])$ are true.

We denote by D_X and D_Y the domains of variables X and Y , respectively.

Plays and Strategies

Let $\mathcal{G} : \langle V, \Theta, \rho_1, \rho_2, \varphi \rangle$ be a game. A **play** of \mathcal{G} is an infinite sequence of states

$$\pi : s_0, s_1, s_2, \dots,$$

satisfying:

- **Initiality:** $s_0 \models \Theta$.
- **Consecution:** For each $j \geq 0$, the state s_{j+1} is a \mathcal{G} -successor of the state s_j .

A play π is said to be **winning for player 2** if $\pi \models \varphi$. Otherwise, it is said to be **winning for player 1**.

A **strategy** for player 1 is a function $\sigma_1 : \Sigma^+ \mapsto D_X$, which determines the next set of values for X following any history $h \in \Sigma^+$. A play $\pi : s_0, s_1, \dots$ is said to be **compatible** with strategy σ_1 if, for every $j \geq 0$, $s_{j+1}[X] = \sigma_1(s_0, \dots, s_j)$.

Strategy σ_1 is **winning** for player 1 from state s if all s -originated plays compatible with σ_1 are winning for player 1. If such a winning strategy exists, we call s a **winning state** for player 1.

Similar definitions hold for player 2 with strategies of the form $\sigma_2 : \Sigma^+ \times D_X \mapsto D_Y$.

From Winning Games to Programs

A game \mathcal{G} is said to be **winning for player 2** (**player 1**, respectively) if **all** (**some**) initial states are winning for **2** (**1**, respectively).

Assume we are given a set of **LTL** specifications. We construct a game as follows:

- As Θ we take all the non-temporal specification parts which relate to the initial state.
- As ρ_1 and ρ_2 , we can take **True**. A more efficient choice is to include in ρ_1 (similarly ρ_2) all local limitations on the next values of X (resp. Y), such as

$$r_i \wedge \neg g_i \rightarrow r'_i$$

- We place in φ all the remaining properties that have not already been included in Θ , ρ_1 , and ρ_2 .

We solve the game, attempting to decide whether the game is winning for player 1 or 2. If it is winning for **player 1** the specification is **unrealizable**. If it is winning for **player 2**, we can extract a winning strategy which is a **working implementation**.

The Game for the **Sample Specification**

For the specification

$$\bigwedge_i (\overline{r}_i \wedge (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \wedge r_i \wedge g_i \Rightarrow \diamond \overline{r}_i) \rightarrow \bigwedge_{i \neq j} \square \neg(g_i \wedge g_j) \wedge \bigwedge_i \left(\overline{g}_i \wedge \begin{pmatrix} r_i = g_i \Rightarrow \bigcirc g_i = g_i \wedge \\ r_i \wedge \overline{g}_i \Rightarrow \diamond g_i \wedge \\ \overline{r}_i \wedge g_i \Rightarrow \diamond \overline{g}_i \end{pmatrix} \right)$$

We take the following game components:

$$X \cup Y : \{r_i \mid i = 1, \dots, n\} \cup \{g_i \mid i = 1, \dots, n\}$$

$$\Theta : \bigwedge_i (\overline{r}_i \wedge \overline{g}_i)$$

$$\rho_1 : \bigwedge_i ((r_i \neq g_i) \rightarrow (r'_i = r_i))$$

$$\rho_2 : \bigwedge_{i \neq j} \neg(g'_i \wedge g'_j) \wedge \bigwedge_i ((r_i = g_i) \rightarrow (g'_i = g_i))$$

$$\varphi : \bigwedge_i (r_i \wedge g_i \Rightarrow \diamond \overline{r}_i) \rightarrow \bigwedge_i ((r_i \wedge \overline{g}_i \Rightarrow \diamond g_i) \wedge (\overline{r}_i \wedge g_i \Rightarrow \diamond \overline{g}_i))$$

Different Solutions to Different Winning Conditions

When applied to controller synthesis, we denote the **controlled predecessor** by $\square p$ with the meaning that $s \models \square p$ iff for every environment (uncontrolled) step leading from s to s' , there exists a system (controlled) successor of s' satisfying p .

Equivalently, s is an $\forall\exists$ -predecessor of p , which can also be written as:

$$\square p : \quad \forall X' : \rho_1(V, X') \rightarrow \exists Y' : \rho_2(V, V') \wedge p(V')$$

With this notation, we can present the following fix-point expressions for computing the winning states corresponding to various winning conditions:

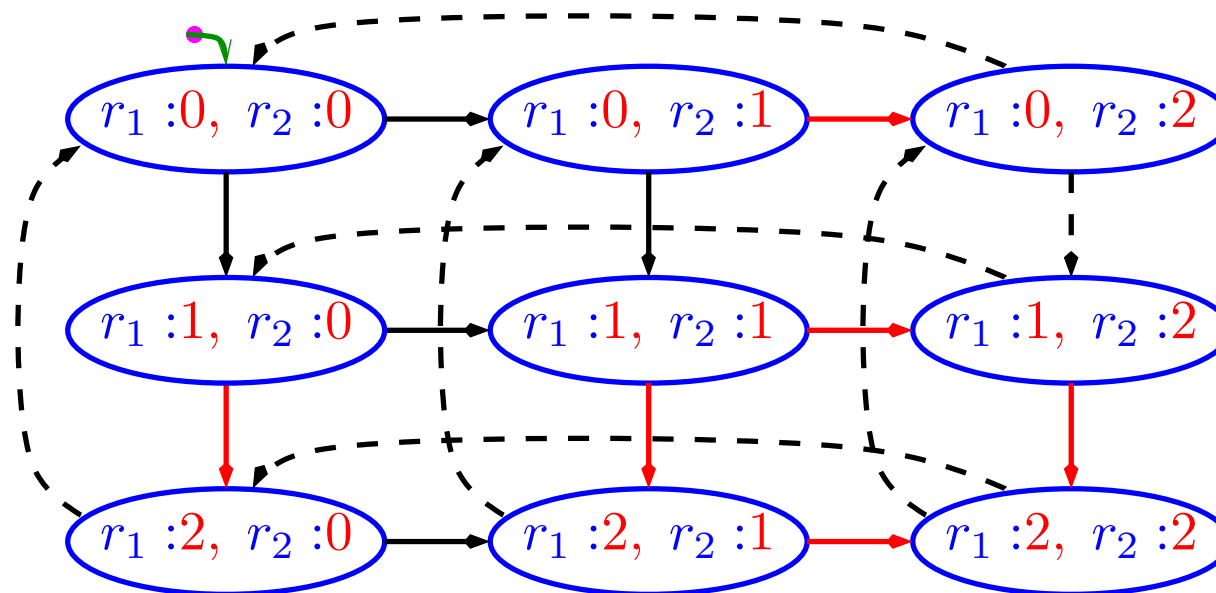
Winning Condition	Fix-point Expression
$\diamond W$	$\mu y. W \vee \square y = W \vee \square W \vee \square \square W \vee \dots$
$\square W$	$\nu y. W \wedge \square y = W \wedge \square W \wedge \square \square W \wedge \dots$
$\square \diamond W$	$\nu z \mu y. W \wedge \square z \vee \square y$

The last case is based on the maximal fix-point solution of the equation

$$z = \mu y. (W \wedge \square z) \vee \square y$$

searching for a visit to a W -state with an enforceable z -successor.

Illustrate on MUX-SEM

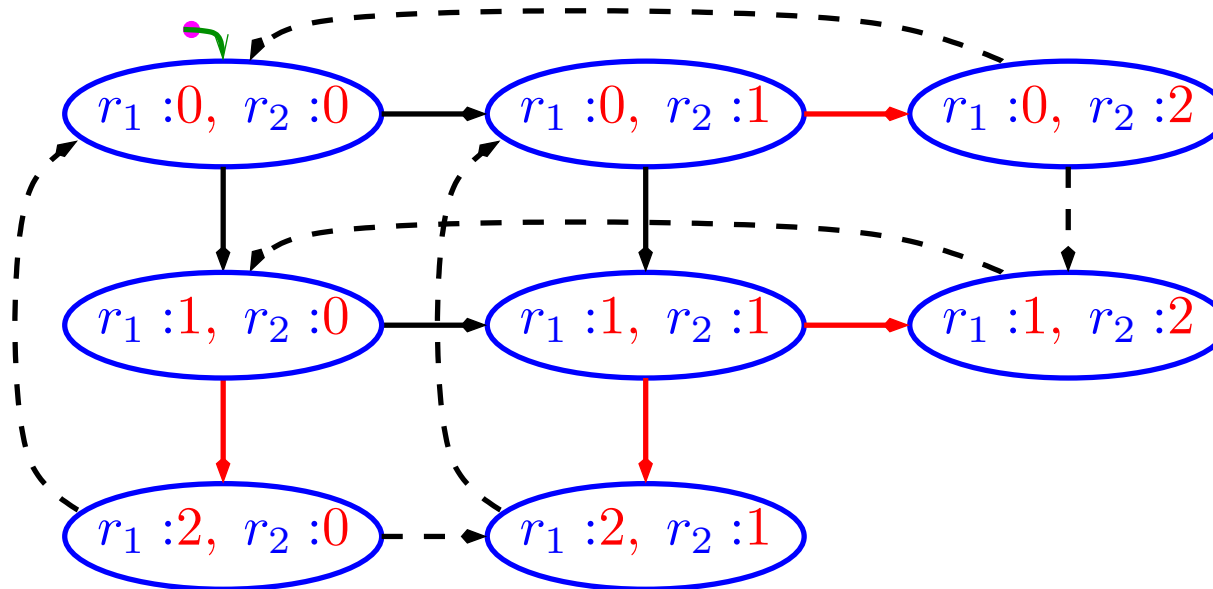


We wish to synthesize a program that guarantees

$$\square \neg(r_1 = 2 \wedge r_2 = 2) \wedge (\square \diamond (r_1 \neq 1) \wedge \square \diamond (r_2 \neq 1))$$

Step 1: Assuring $\square \neg(r_1 = 2 \wedge r_2 = 2)$

Applying the synthesis algorithm for this formula, we obtain



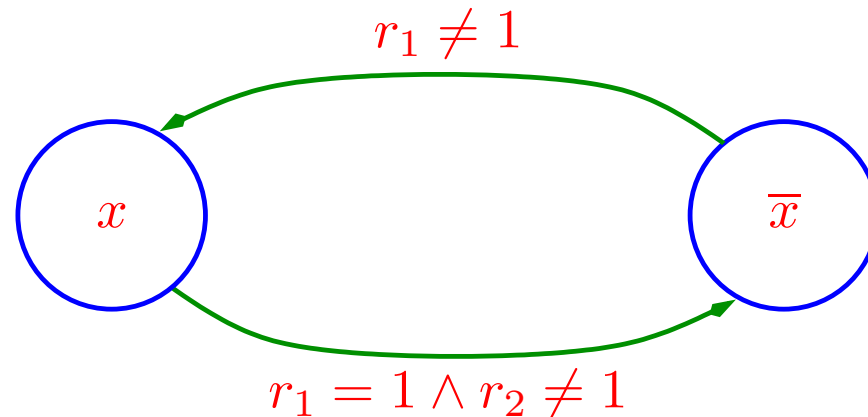
Have still to satisfy

$$(\square \diamond (r_1 \neq 1) \wedge \square \diamond (r_2 \neq 1))$$

which is not of the form guaranteeing a memory-less strategy.

From Multi-Recurrence to Simple Recurrence

We can construct a (deterministic) automaton (equivalently an **FDS**) which monitors for alternating occurrences of $r_1 \neq 1$ and $r_2 \neq 1$.



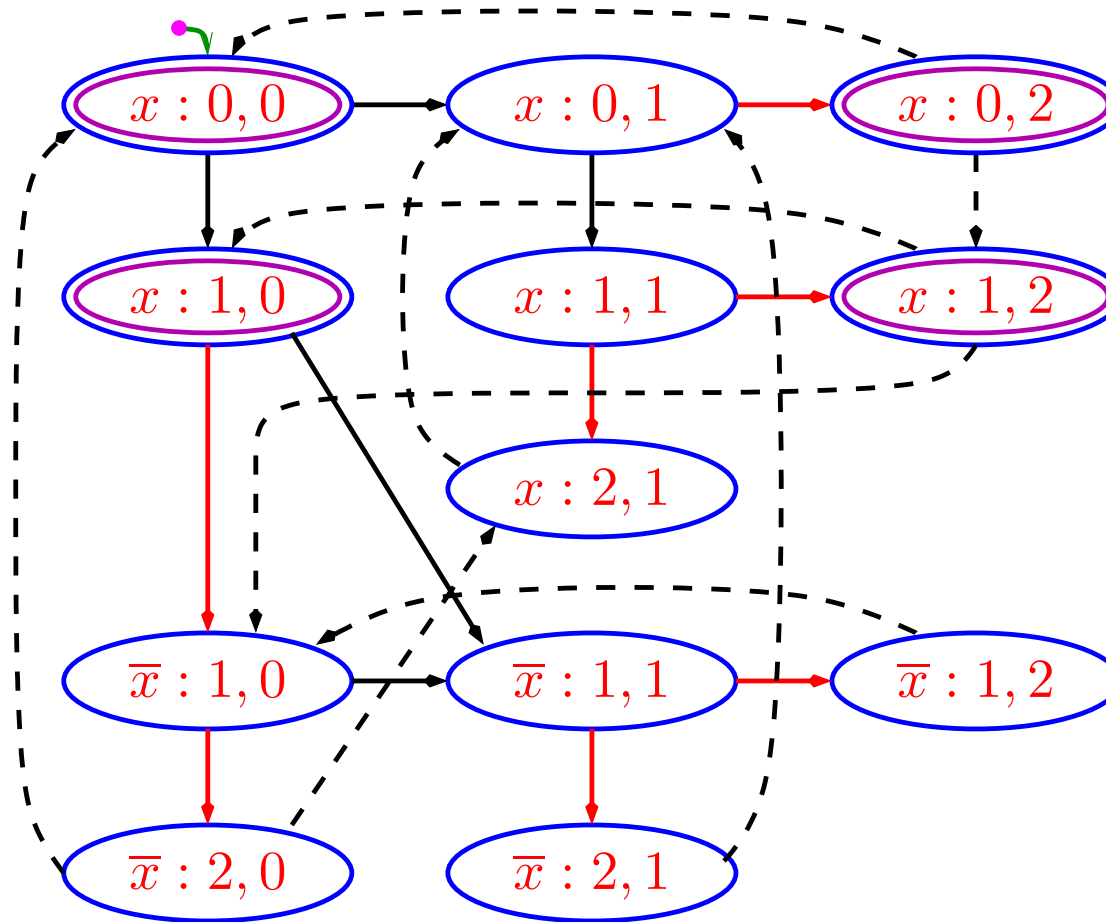
This automaton can be defined as an **FDS** **A** with the transition relation:

$$x' = r_1 \neq 1 \vee x \wedge r_2 = 1$$

It can be shown that $\square \diamond (x \wedge r_2 \neq 1)$ iff $\square \diamond (r_1 \neq 1) \wedge \square \diamond (r_2 \neq 1)$.

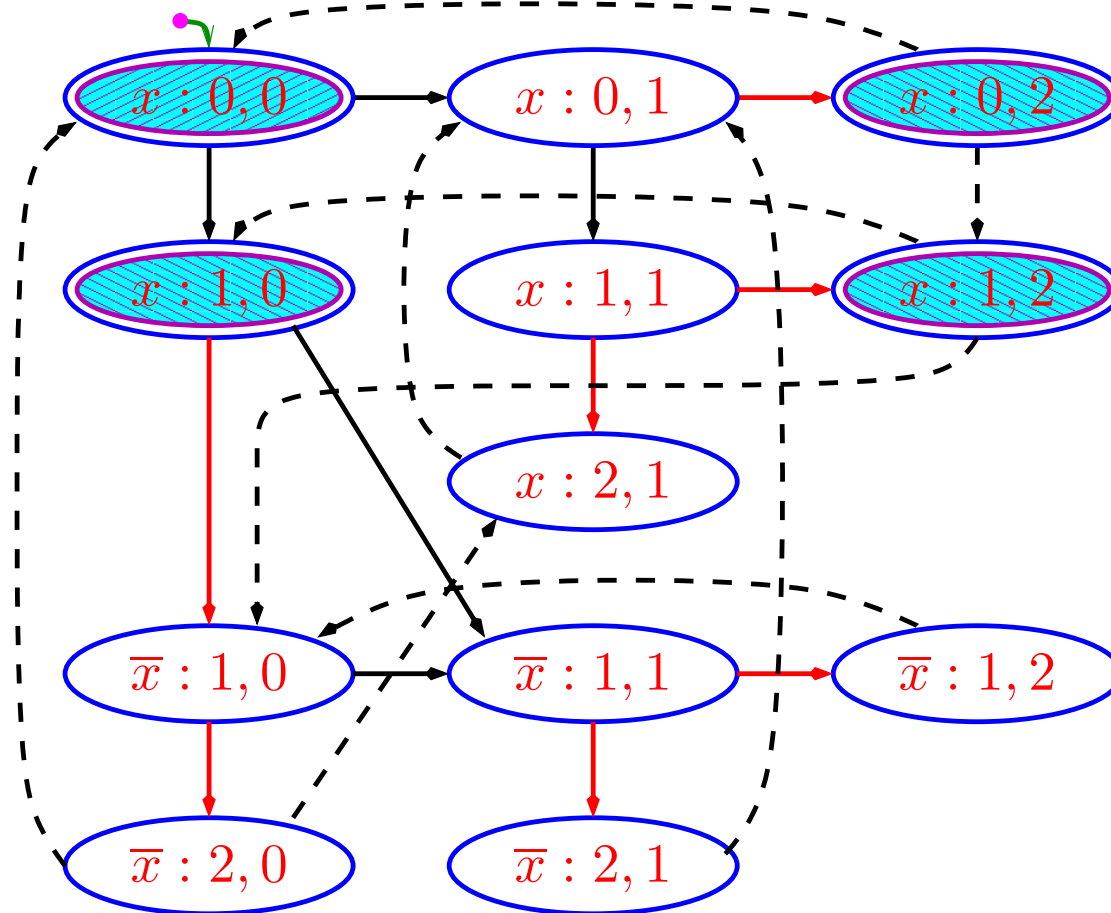
Form the Parallel Composition and Solve

We can now form the parallel composition of the system and the FDS A , and solve for the winning condition $\square \lozenge (x \wedge r_2 \neq 1)$.

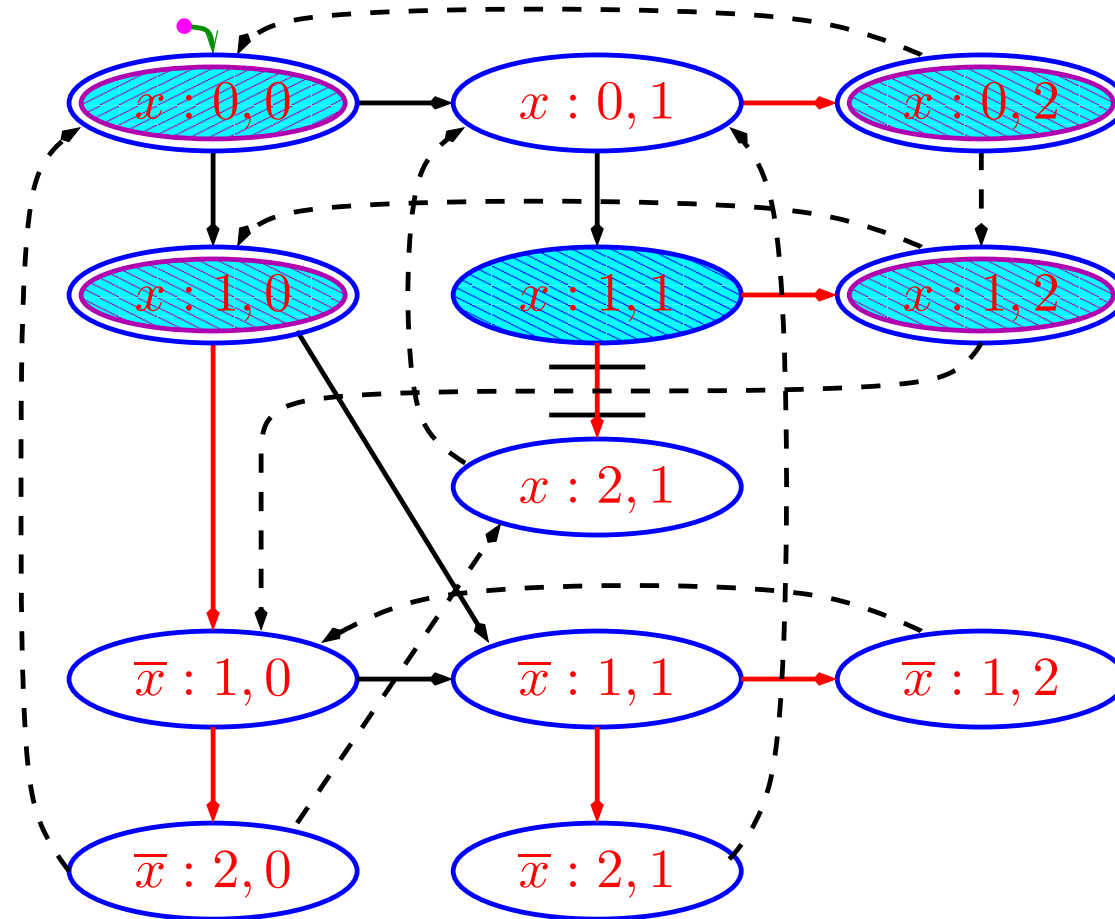


Solving: Step 0

Mark all immediately winning States as members of .

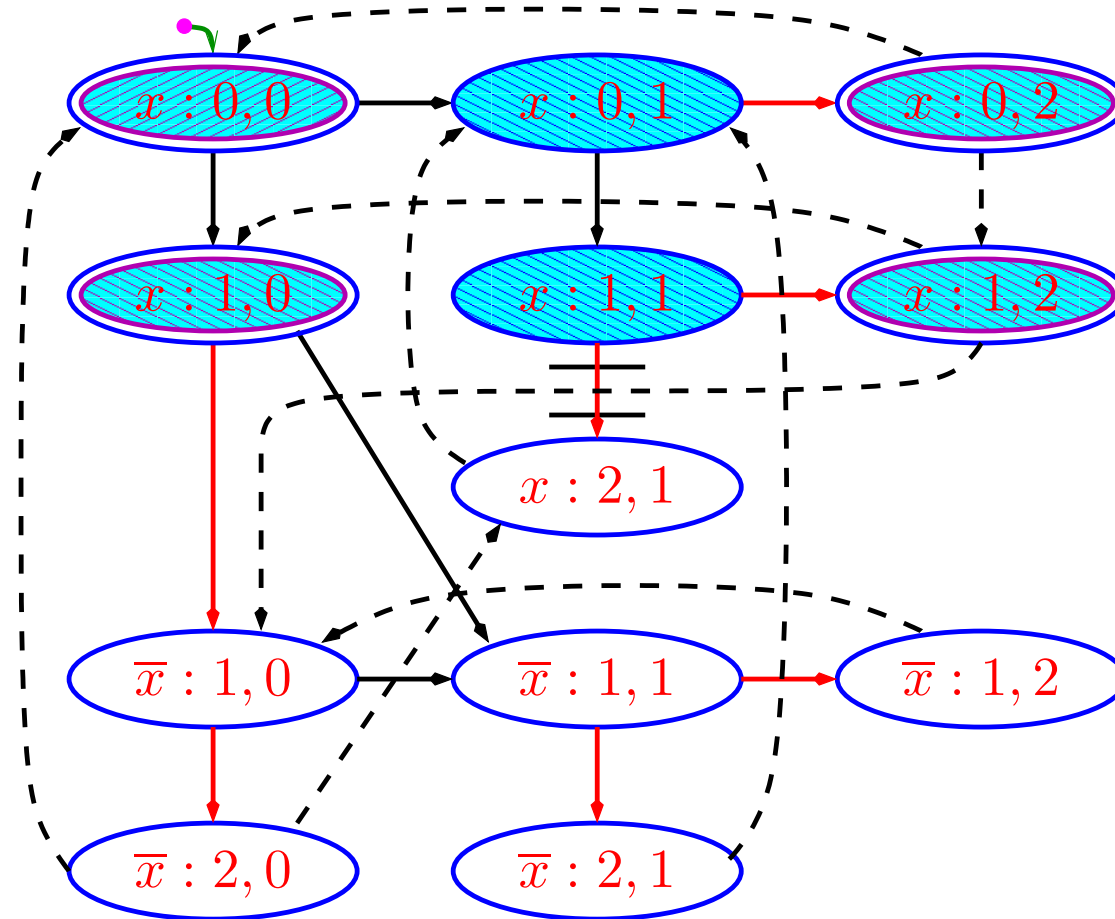


Solving: Step 1



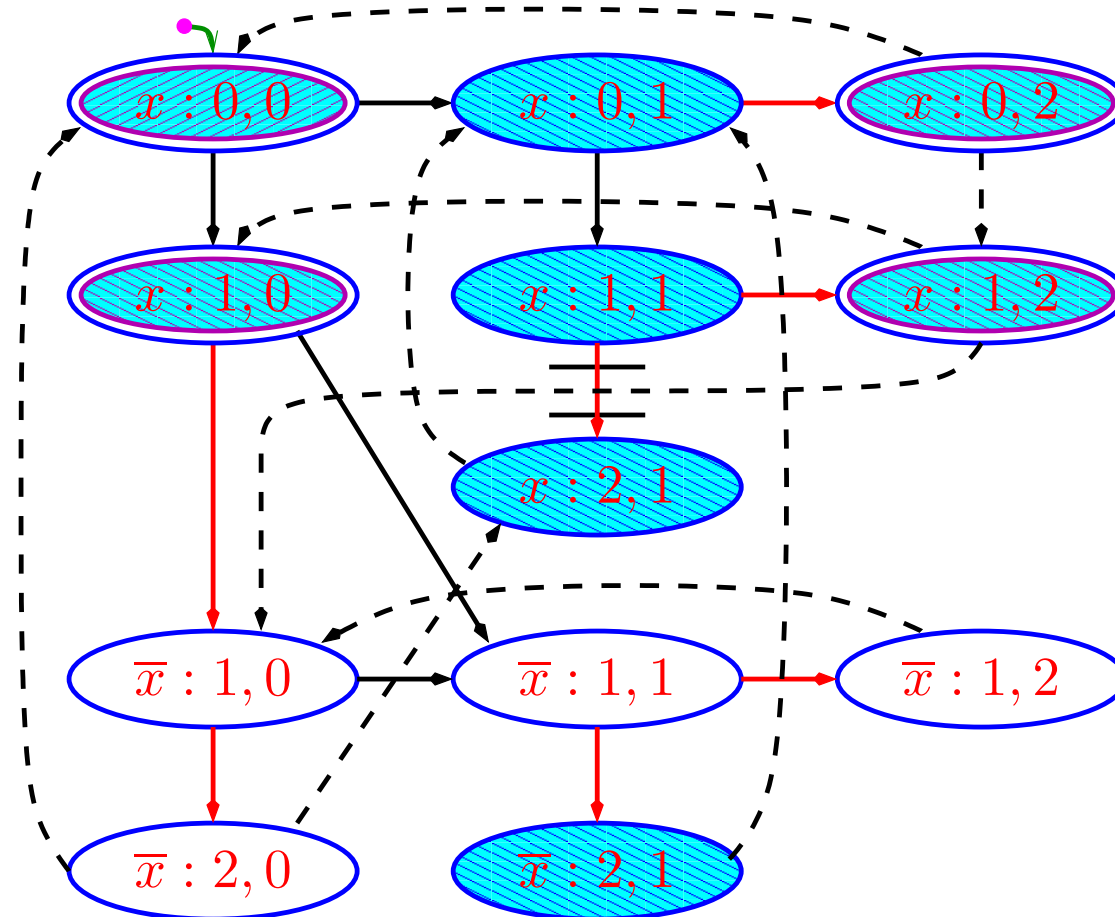
Add state $(x : 1, 1)$ since it has a winning successor.

Solving: Step 2



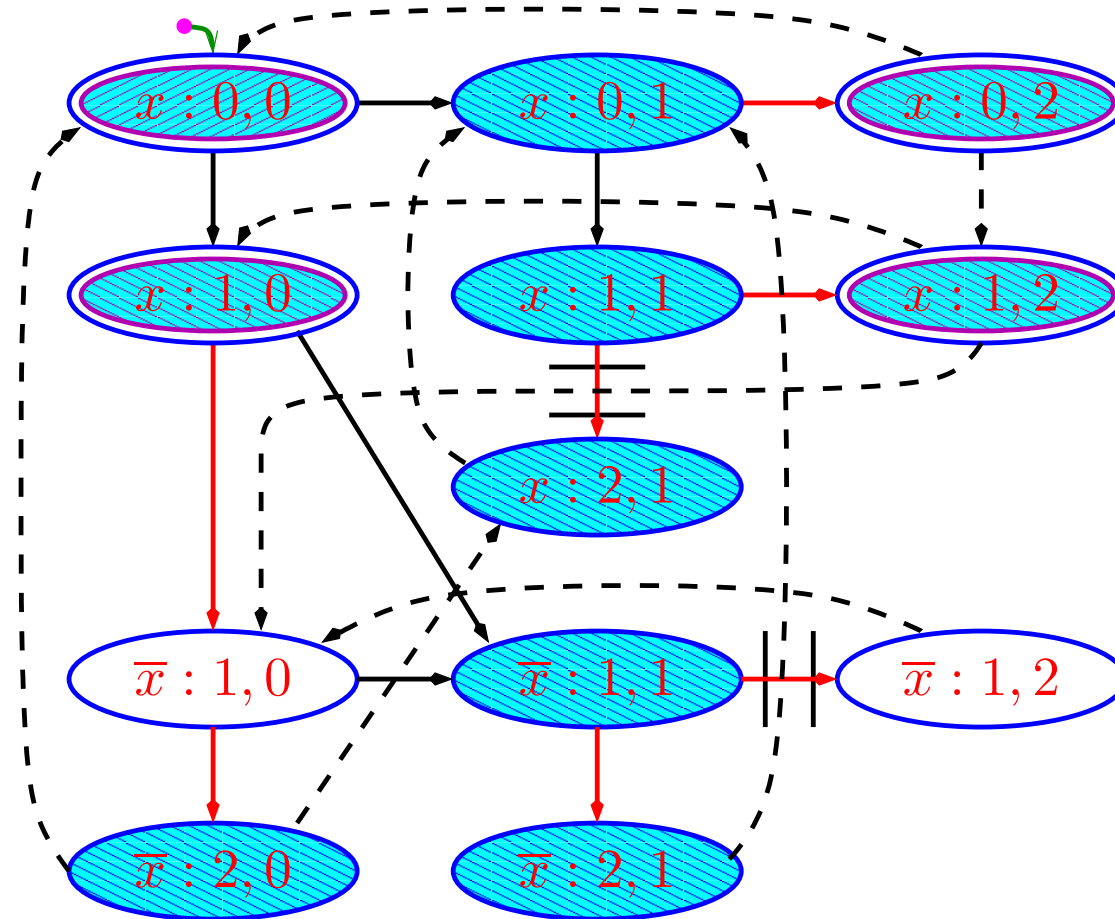
Add state $(x : 0, 1)$ since it has a winning successor.

Solving: Step 3



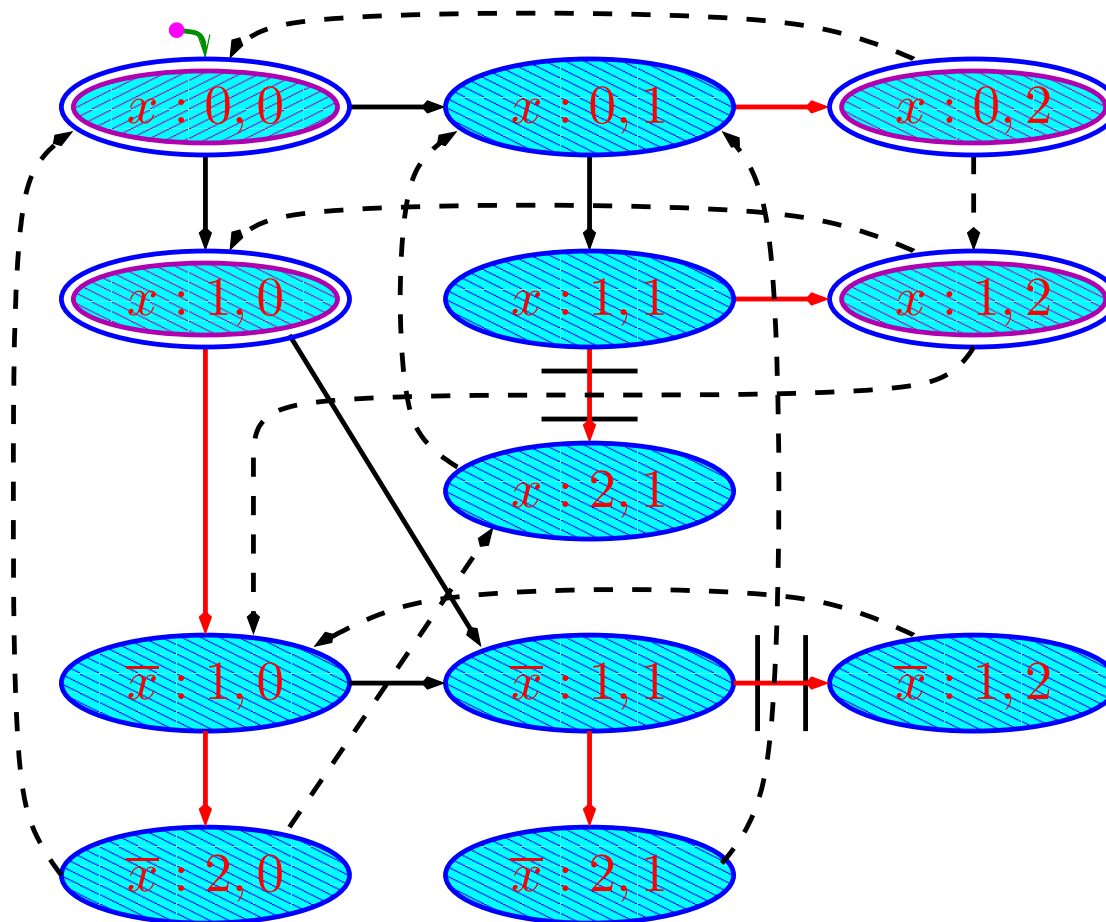
Add states $(x : 2, 1)$ and $(\bar{x} : 2, 1)$ since they each have only winning successors.

Solving: Step 4



Add state $(x : 2, 0)$ which has only winning successors. Also add $(\bar{x} : 1, 1)$ since it has one winning successor. Choose $(\bar{x} : 2, 1)$ to be the strategic successor of $(\bar{x} : 1, 1)$.

Solving: Step 5



Add state $(\bar{x} : 1, 0)$ all of whose successors are winning. Then add $(\bar{x} : 1, 2)$. This concludes the first iteration and also the full computation.

Note the ultimately periodic sequence:

$$(x : 0, 0), [(x : 0, 1), (x : 1, 1), (x : 1, 2), (\bar{x} : 1, 0), (\bar{x} : 1, 1), (\bar{x} : 2, 1)]^*$$

Simple Reactivity[1] Games

When moving from specification $\square \diamond q$ to $(\square \diamond p \rightarrow \square \diamond q)$, the fix-point expression extends from

$$\nu Z \mu Y. (q \wedge \square Z) \vee \square Y$$

to

$$\nu Z \mu Y \nu X. (q \wedge \square Z) \vee \square Y \vee (\neg p \wedge \square X)$$

Namely, in the progress towards the next q visit, we can either take a step which moves us closer to q , or stay a while (possibly forever) in states satisfying $\neg p$.

Solving Games for Generalized **Reactivity[1]** (**Streett[1]**)

Following [KPP03], we present an n^3 algorithm for solving games whose winning condition is given by the (generalized) **Reactivity[1]** condition

$$\diamond \square p_1 \vee \diamond \square p_2 \vee \cdots \vee \diamond \square p_m \vee \square \diamond q_1 \wedge \square \diamond q_2 \wedge \cdots \wedge \square \diamond q_n$$

equivalently,

$$(\square \diamond p_1 \wedge \square \diamond p_2 \wedge \cdots \wedge \square \diamond p_m) \rightarrow \square \diamond q_1 \wedge \square \diamond q_2 \wedge \cdots \wedge \square \diamond q_n$$

This class of properties is bigger than the properties specifiable by deterministic **Büchi** automata. It covers a great majority of the properties we have seen in the **Prosyd** project so far.

For example, a specification for an **arbiter** system will be of the form

$$(\cdots \wedge g_i \Rightarrow \diamond \neg r_i \wedge \cdots) \rightarrow \cdots \wedge r_i \Rightarrow \diamond g_i \wedge \cdots$$

Response vs. Recurrence Properties

Every response formula $p \Rightarrow \diamond q$ is equivalent to a recurrence formula $\square \diamond r$ for some **past** formula r . This is because

$$p \Rightarrow \diamond q \quad \sim \quad \square \diamond ((\neg p) \mathcal{B} q)$$

For the case of the **Arbiter** specification, such conversion is not necessary, because we can rewrite the liveness requirements as follows:

Rewrite $r_i \wedge g_i \Rightarrow \diamond \bar{r}_i$ as $\square \diamond \neg(r_i \wedge g_i)$

Rewrite $r_i \wedge \bar{g}_i \Rightarrow \diamond g_i$ and $\bar{r}_i \wedge g_i \Rightarrow \diamond \bar{g}_i$ as $\square \diamond (r_i = g_i)$

The Solution

The winning states in a **Streett[1]** game can be computed by

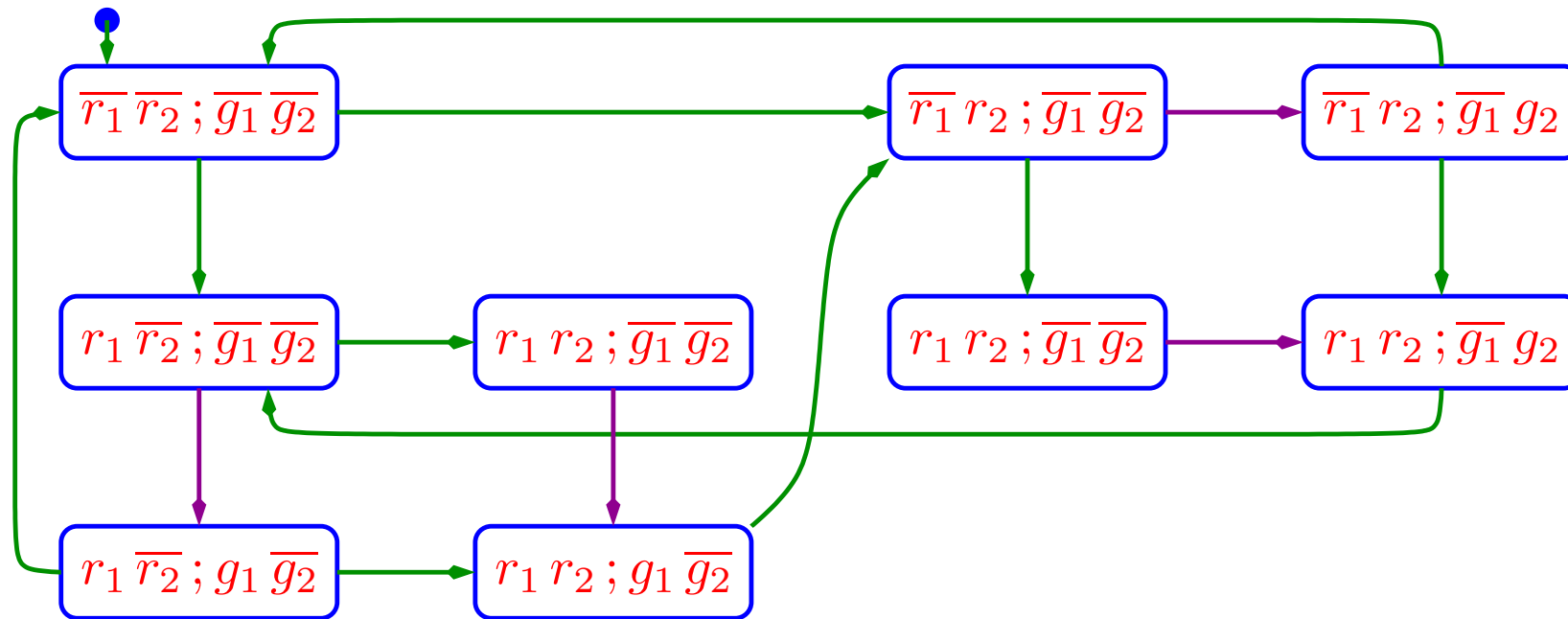
$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{j=1}^m \nu X (q_1 \wedge \boxed{\diamond} Z_2 \vee \boxed{\diamond} Y \vee \neg p_j \wedge \boxed{\diamond} X) \right) \\ \mu Y \left(\bigvee_{j=1}^m \nu X (q_2 \wedge \boxed{\diamond} Z_3 \vee \boxed{\diamond} Y \vee \neg p_j \wedge \boxed{\diamond} X) \right) \\ \vdots \\ \mu Y \left(\bigvee_{j=1}^m \nu X (q_n \wedge \boxed{\diamond} Z_1 \vee \boxed{\diamond} Y \vee \neg p_j \wedge \boxed{\diamond} X) \right) \end{bmatrix}$$

where

$$\boxed{\diamond} \varphi : \forall X' : \rho_1(V, X') \rightarrow \exists Y' : \rho_2(V, V') \wedge \varphi(V')$$

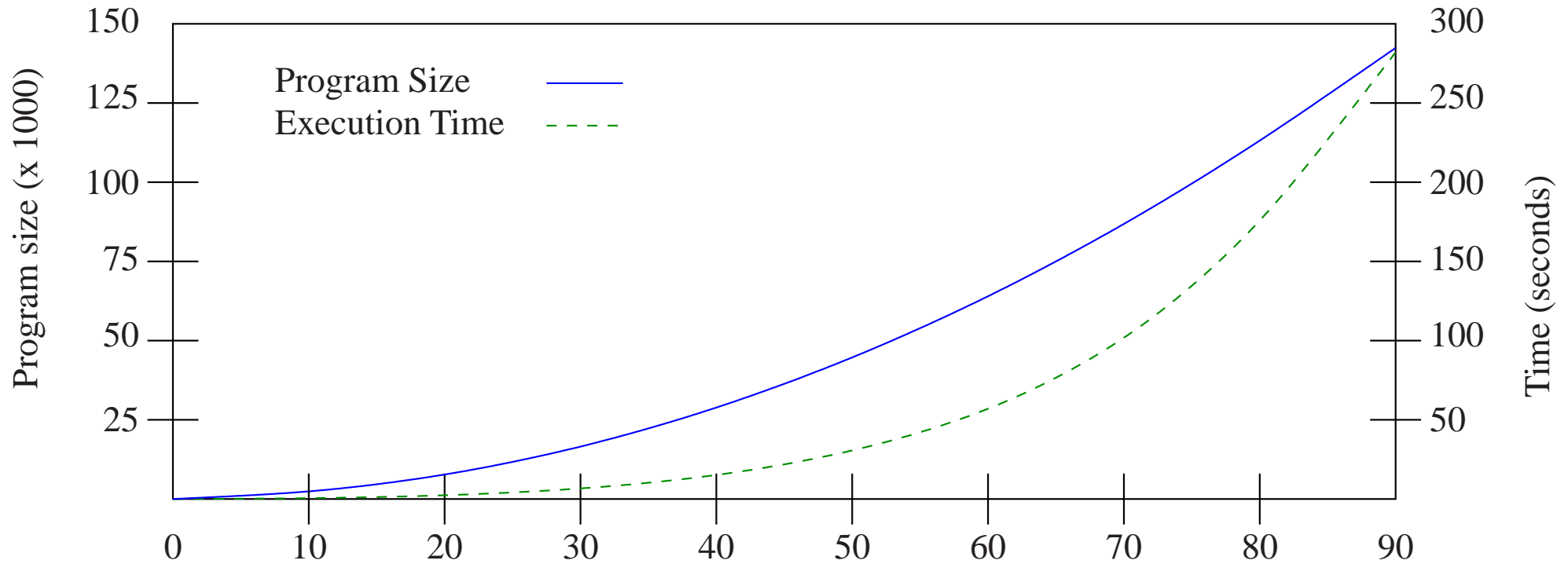
Results of Synthesis

The design realizing the specification can be extracted as the winning strategy for Player 2. Applying this to the **Arbiter** specification, we obtain the following design:



We have a symbolic algorithm for extracting the implementing design/winning strategy.

Execution Times and Programs Size for Arbitrator(N)



Extent of Properties Class

The presented algorithm is applicable to all properties which can be specified by a formula of the form

$$(\varphi_1 \wedge \cdots \wedge \varphi_m) \rightarrow \psi_1 \wedge \cdots \wedge \psi_n$$

where each φ_i, ψ_i can be specified by a deterministic Büchi automaton.

For example, the LTL formula $\psi_j : p \Rightarrow \diamond q$ can be specified by the deterministic Büchi automata, whose transition relation is given by:

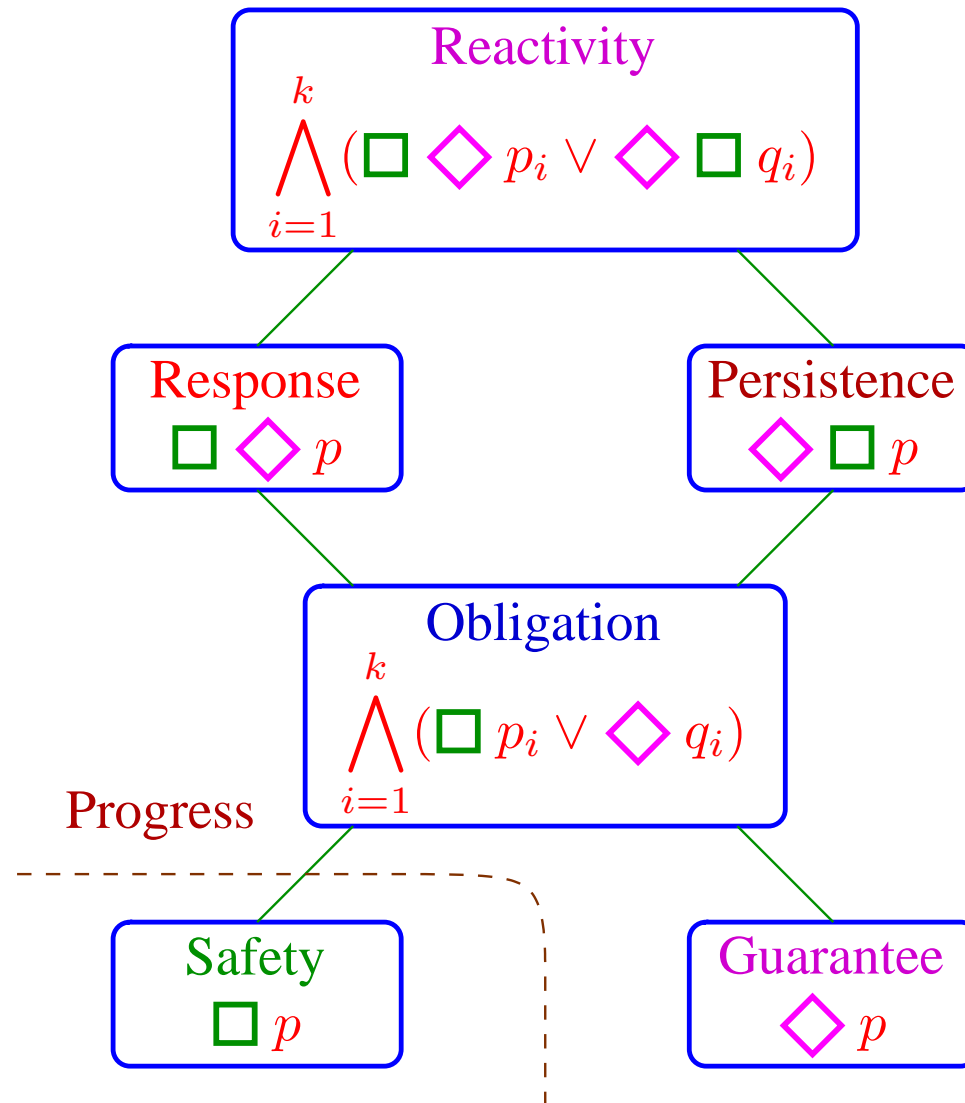
$$x' = (q \vee x \wedge \neg p)$$

Thus, we can add this transition relation to ρ_2 , and replace ψ_j by $\square \diamond x$.

Conclusions

- It is possible to perform design synthesis for restricted fragments of **LTL** in acceptable time.
- The tractable fragment (**Street(1)**) covers most of the properties that appear in standard specifications.
- It is worthwhile to invest an effort in locating the formula within the temporal hierarchy. Solving a game in **React(k)** has complexity $N^{(2k+1)}$.

Hierarchy of the Temporal Properties



where p , p_i , q , q_i are **past** formulas. A unique proof rule was developed for each of the classes.