Synthesis of Designs from Temporal Specifications

Amir Pnueli

New York University and Weizmann Institute of Sciences

IBM Verification Conference, November 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?

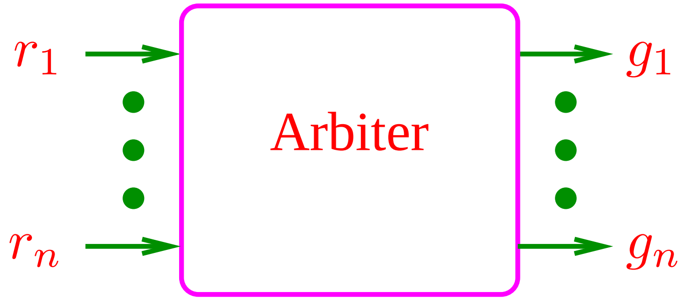
A Brief History of System Synthesis

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

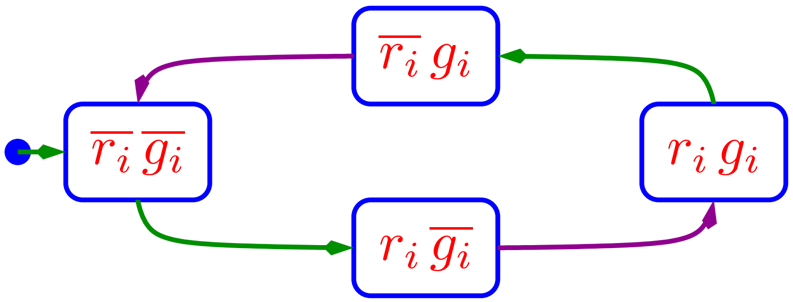
- Determine if there exists an automaton (sequential circuit) which realizes the specification.
- If the specification is realizable, construct an implementing circuit

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

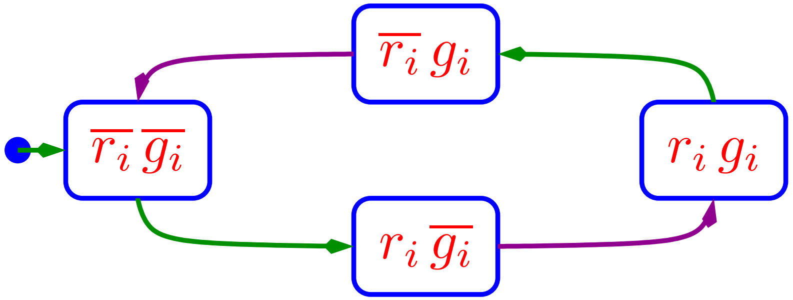
Example of a Specification: Arbiter



The protocol for each client:



The Behavioral Specification



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

$$\bigwedge_{i \neq j} \forall t : \neg g_i[t] \vee \neg g_j[t] \quad \wedge$$

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

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.

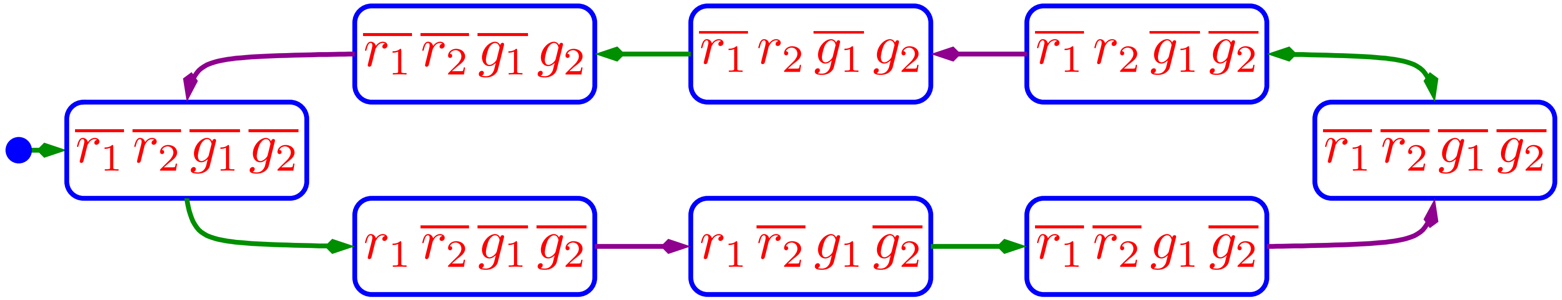
At the same year, Büchi and Landweber provided another solution, based on infinite games.

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:



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 feasible.

Inconsistency

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

Unrealizability For a system

Realizing the specification

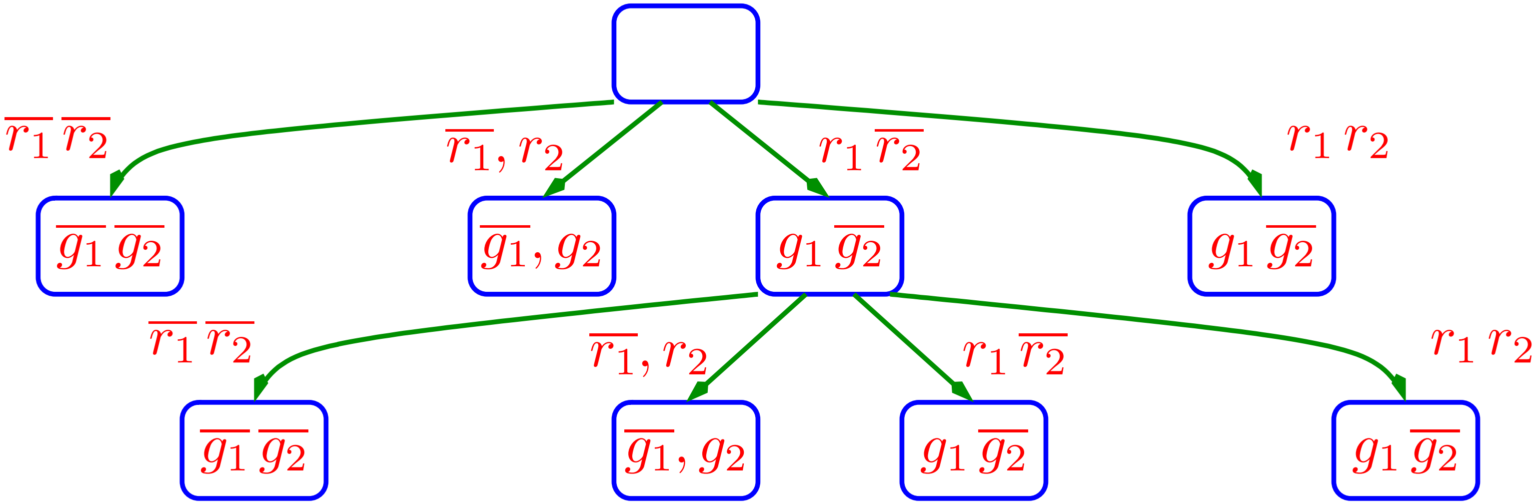
$$g \quad \longleftrightarrow \quad \Diamond\, r$$

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) 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

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

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.

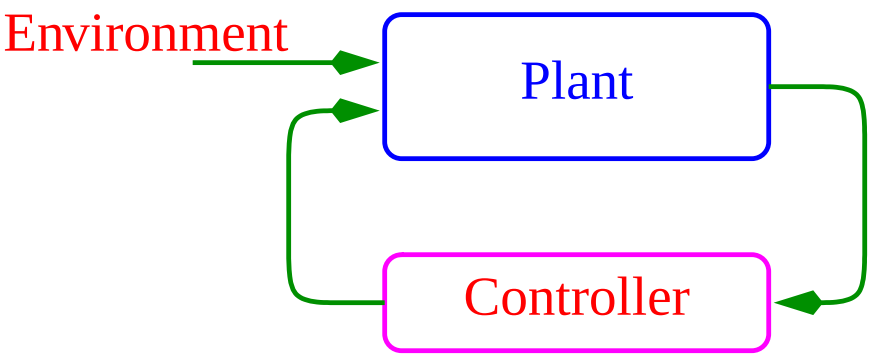
Simple Cases of Lower Complexity

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.

The Control Framework

Classical (Continuous Time) Control



Required: A design for a controller which will cause the plant to behave correctly under all possible (appropriately constrained) environments.

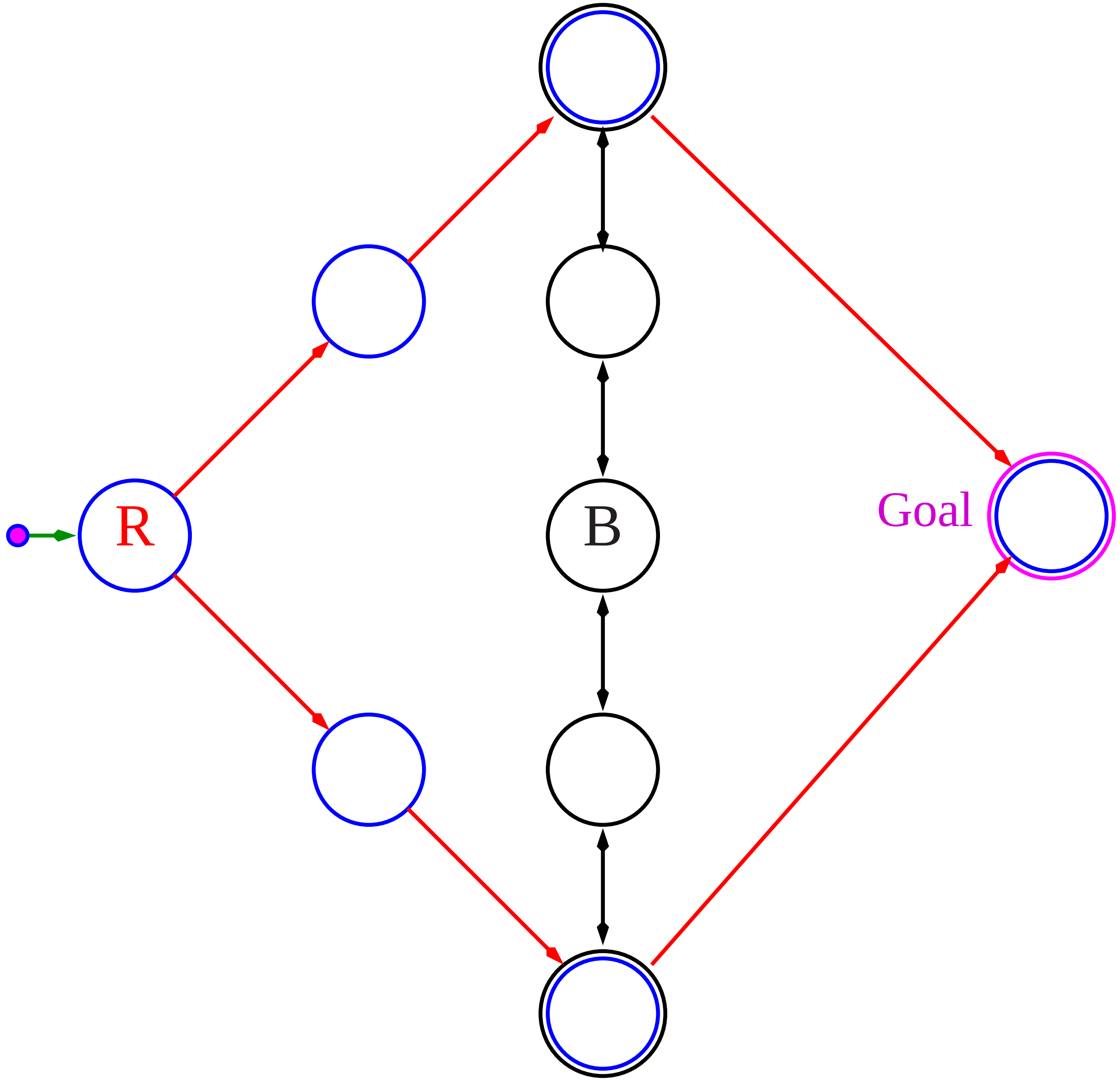
Discrete Event Systems Controller: [Ramadge and Wonham 89]. Given a Plant which describes the possible events and actions. Some of the actions are controllable while the others are not.

Required: Find a strategy for the controllable actions which will maintain a correct behavior against all possible adversary moves. The strategy is obtained by pruning some controllable transitions.

Application to Reactive Module Synthesis: [PR88], [ALW89] — The Plant represents all possible actions. Module actions are controllable. Environment actions are uncontrollable.

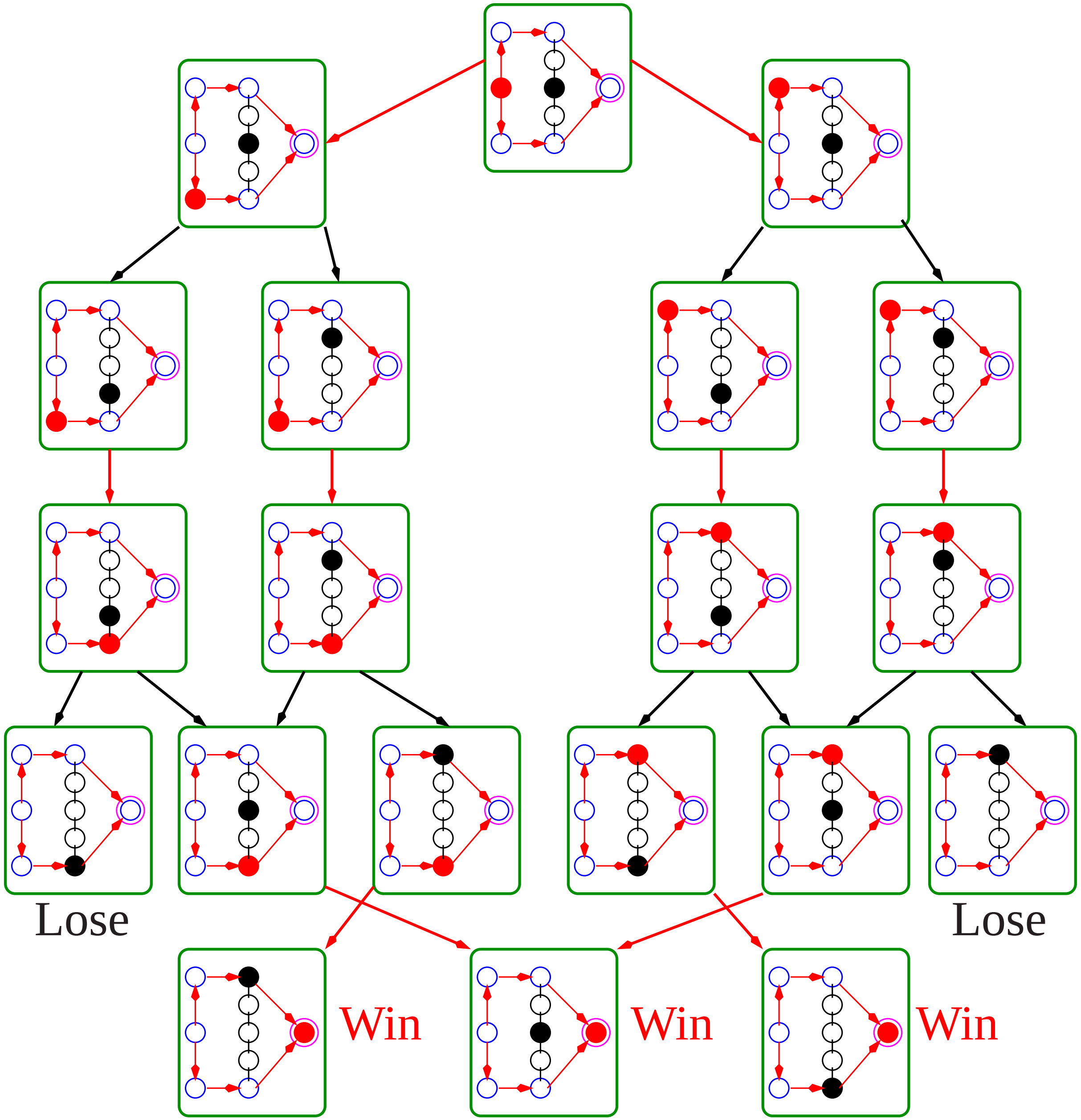
Required: Find a strategy for the controllable actions which will maintain a temporal specification against all possible adversary moves. Derive a program from this strategy. View as a two-persons game.

The Runner Blocker System



Runner R tries to reach the goal. Blocker B tries to intercept and stop R.

State Transitions Diagram



Is the Goal Reachable?

All of our algorithms will be computing sets of states out of the state-transition diagram. Let $\|win\|$ denote the set of states labeled by the *win* proposition. Let ρ be the transition relation, such that $\rho(s_1, s_2)$ holds whenever s_2 is a direct successor of the state s_1 in the state-transition diagram.

For a state-set S, we introduce the predecessor operator $Pre_\exists$ which computes the set of all one-step predecessors of the states in S. That is,

$$Pre_\exists(S) \quad = \quad \{s \mid s \text{ has a } \rho\text{-successor in } S\}$$

Recursively, we define a state s to be goal reaching if either $s \in \|win\|$ or s has a goal reaching successor. That is,

$$R = \|win\| \ \cup \ Pre_\exists(R)$$

We may expect that the solution to this fix-point equation, will give us the set of all states from which $\|win\|$ is reachable.

Among the Possible Solutions, Pick the Minimal

We should take the minimal solution of the fix-point equation $R = \|win\| \cup Pre_{\exists}(R)$ which we denote by

$$\mu R. (\|win\| \ \cup \ Pre_{\exists} \ R)$$

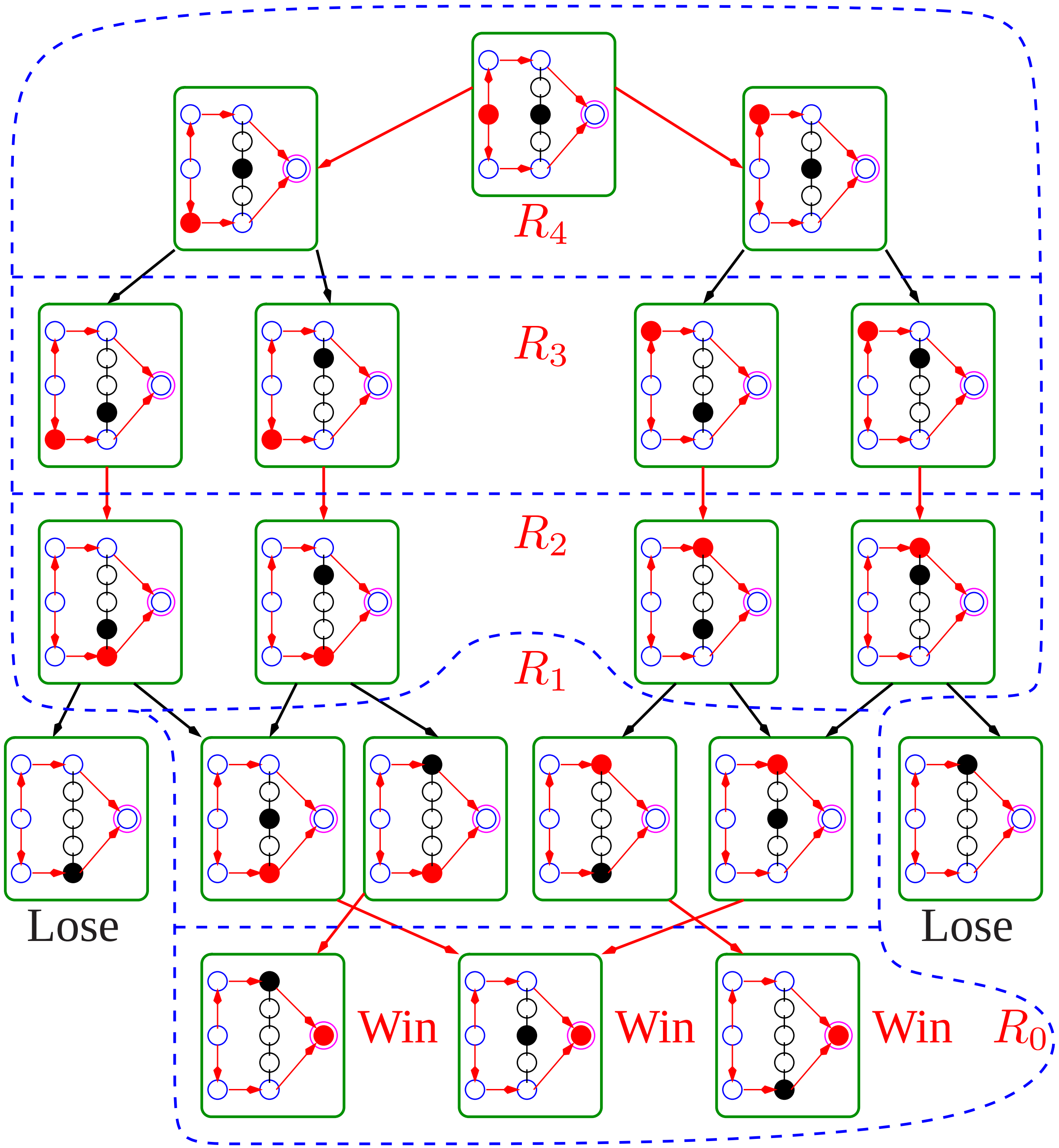
This minimal solution can be effectively computed by the iteration sequence:

$$\begin{array}{rcl} R_0 & = & \emptyset \\ R_1 & = & \|win\| \\ R_2 & = & R_0 \ \cup \ Pre_{\exists} \ R_0 \\ R_3 & = & R_1 \ \cup \ Pre_{\exists} \ R_1 \\ & & \quad \cdots \end{array}$$

Consequently, the goal is reachable from an initial state s_0 iff

$$s_0 \in \mu R. (\|win\| \ \cup \ Pre_{\exists} \ R)$$

Computing $\mu R.\ \|win\| \cup Pre_{\exists}(R)$



Controller Synthesis

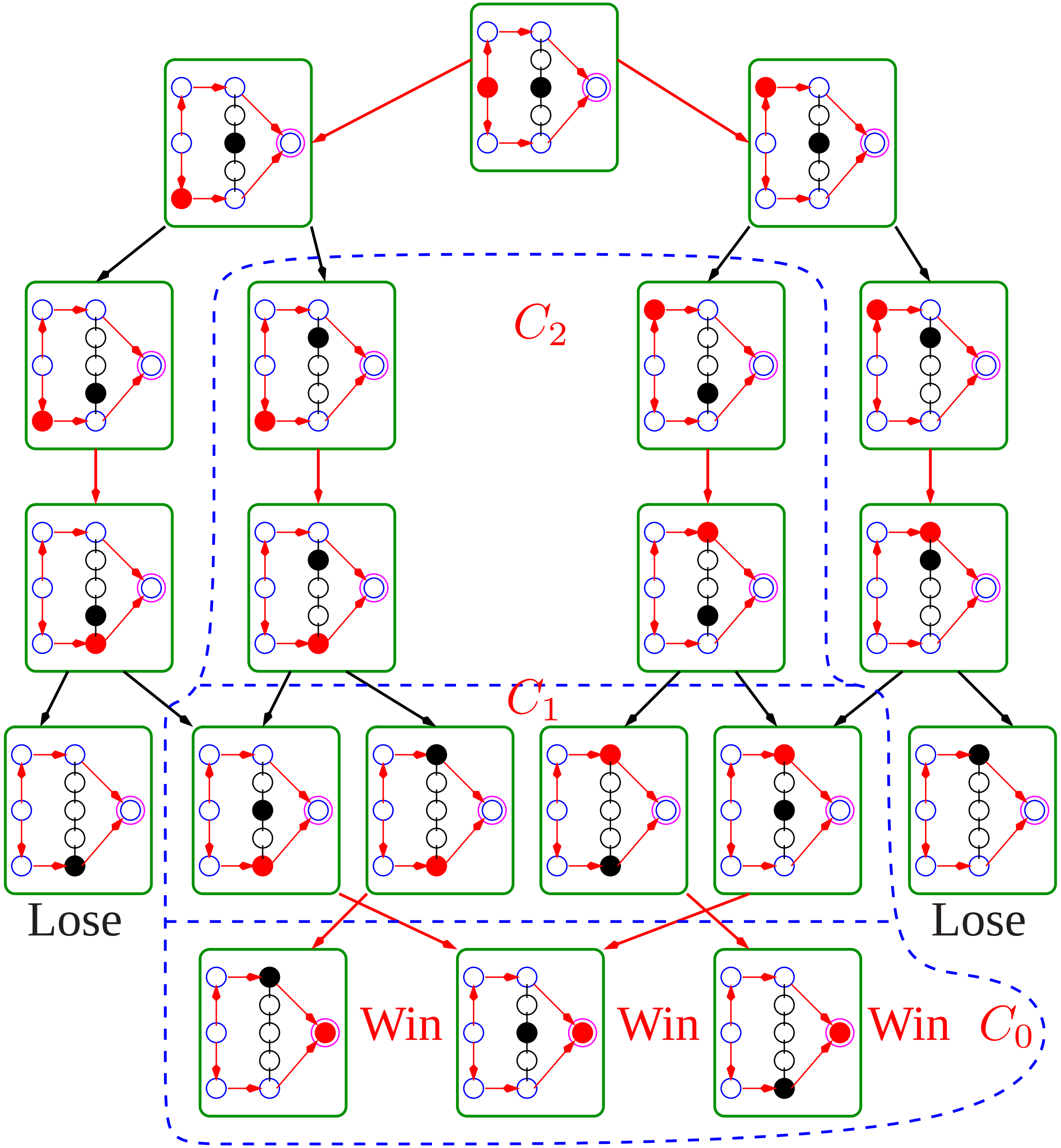
For a set of states C, we define the operator $Pre_{\forall}$ which is dual to $Pre_{\exists}$ and can be defined by

$$Pre_{\forall}(C) \quad = \quad \{s \mid \text{All the } \rho\text{-successors of } s \text{ are in } C\}$$

The two operators can be combined, and the expression $Pre_{\exists\forall}(C) = Pre_{\exists}(Pre_{\forall}(C))$ denotes the set of states s which have at least one successor s_1 all of whose successors belong to C. If we think about the moves as taken in turn by two players, then $Pre_{\exists\forall}(C)$ denotes the states from which the first player can force the game after a complete round (each player making one move) into a C-state.

The expression $control(win) = \mu C. \|\text{win}\| \cup Pre_{\exists\forall}(C)$ characterizes all the states from which a win can be enforced in a finite number of moves.

Computing $\mu C. \|win\| \cup Pre_{\exists\forall}(C)$

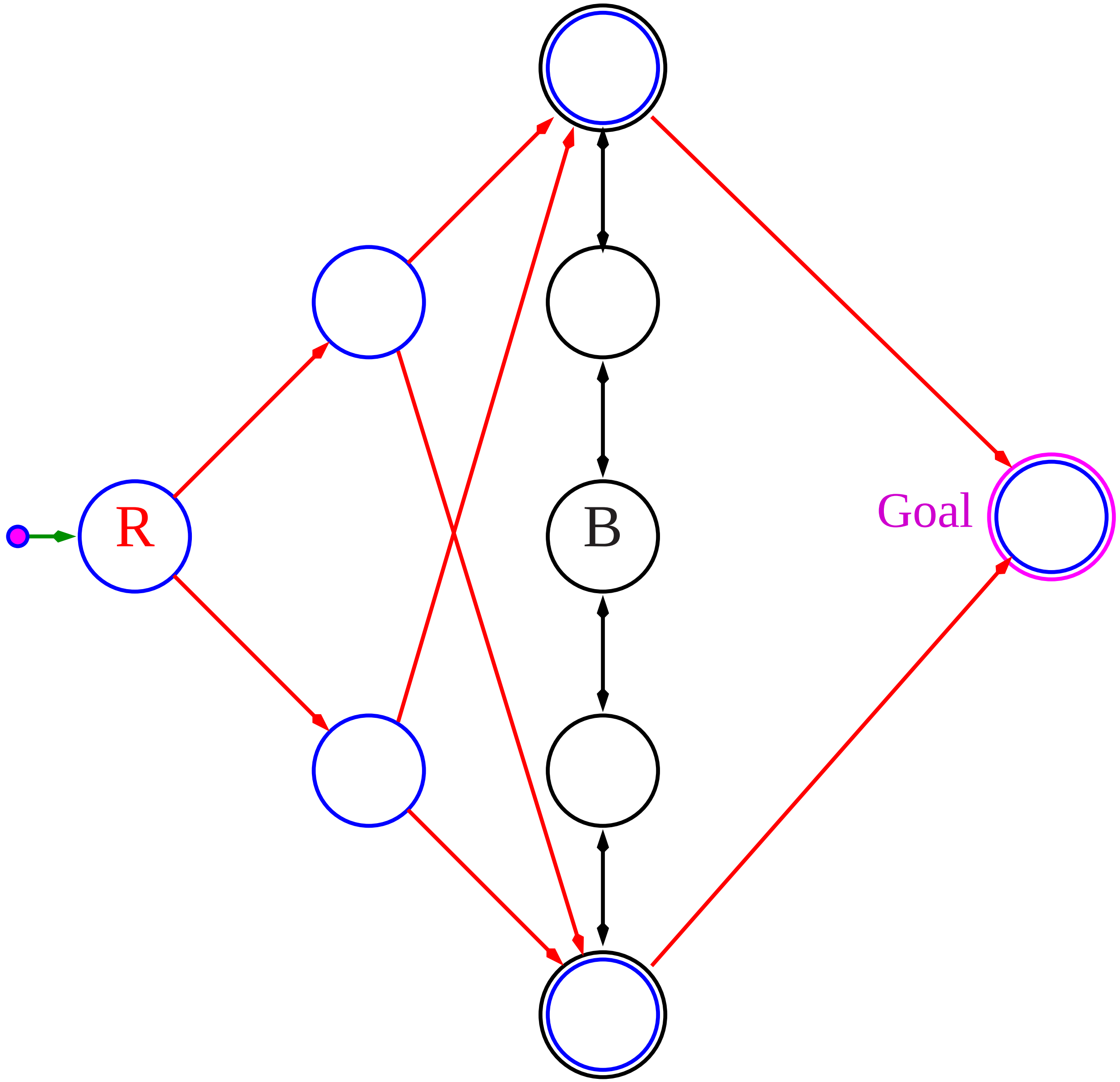


Local Conclusions

The runner and the blocker can cooperate to reach a winning state for R.

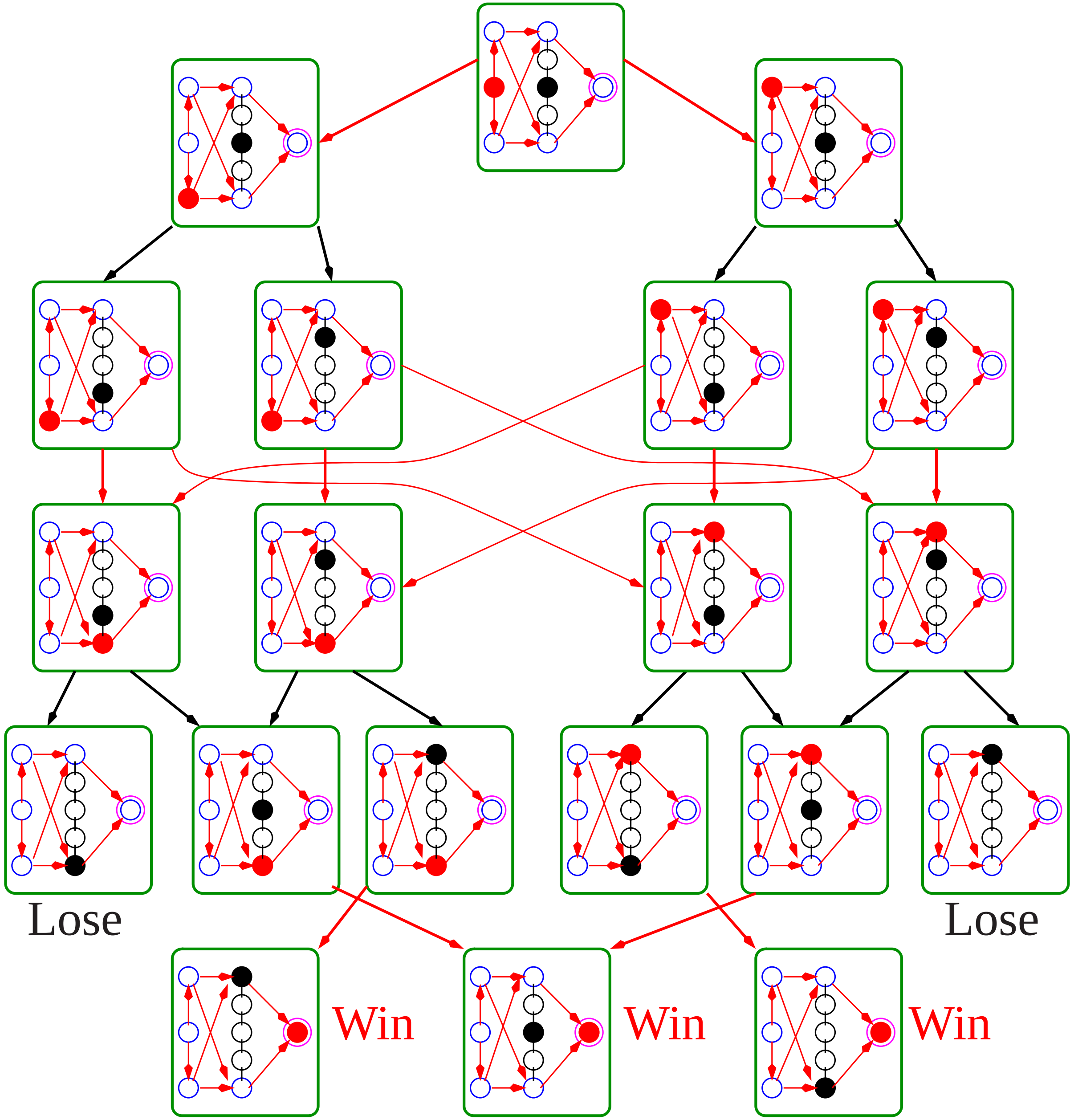
However, R cannot force a win.

A Modified Runner Blocker System

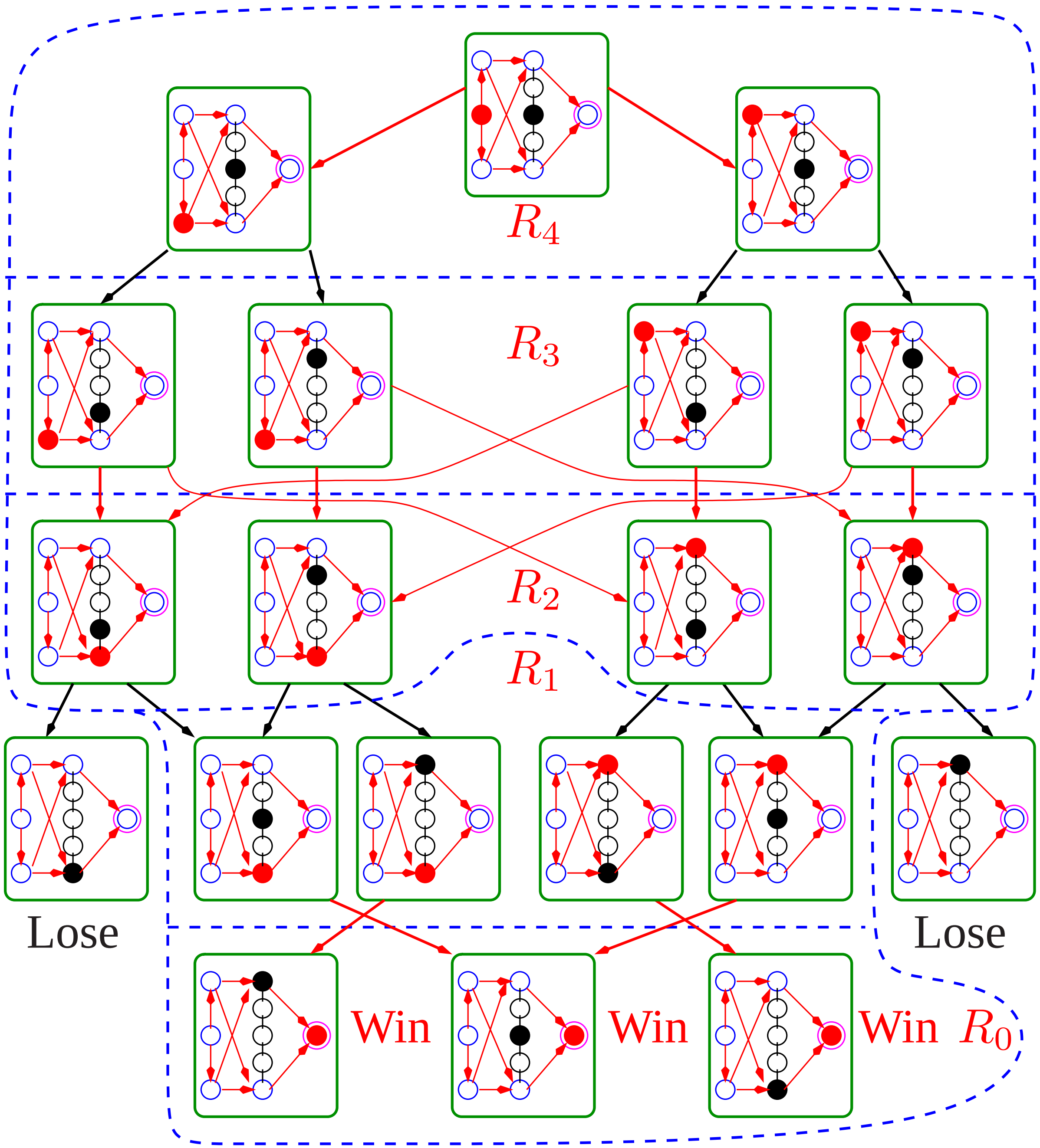


Additional transitions have been added to the runner.

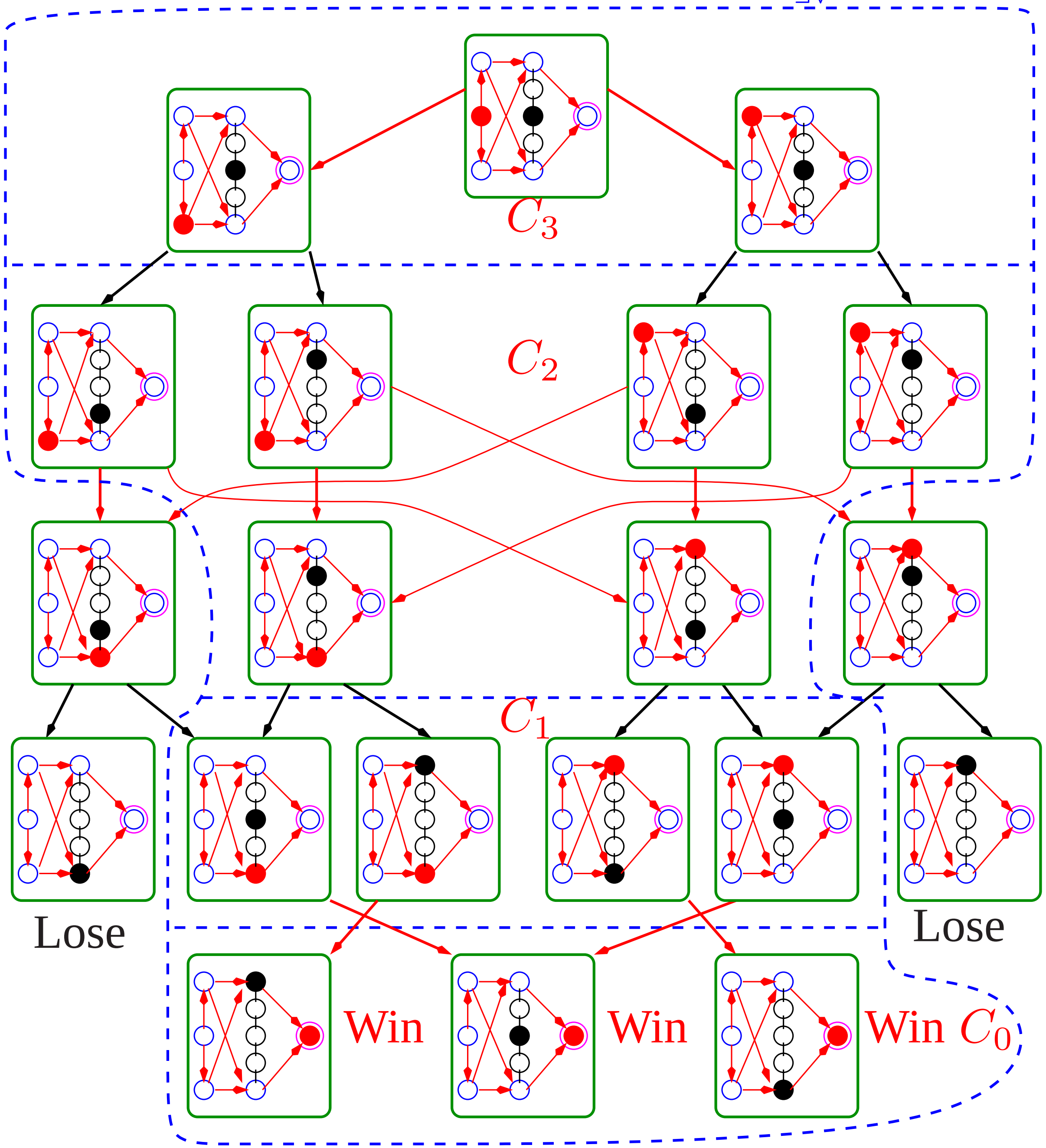
Game Tree for the Modified System



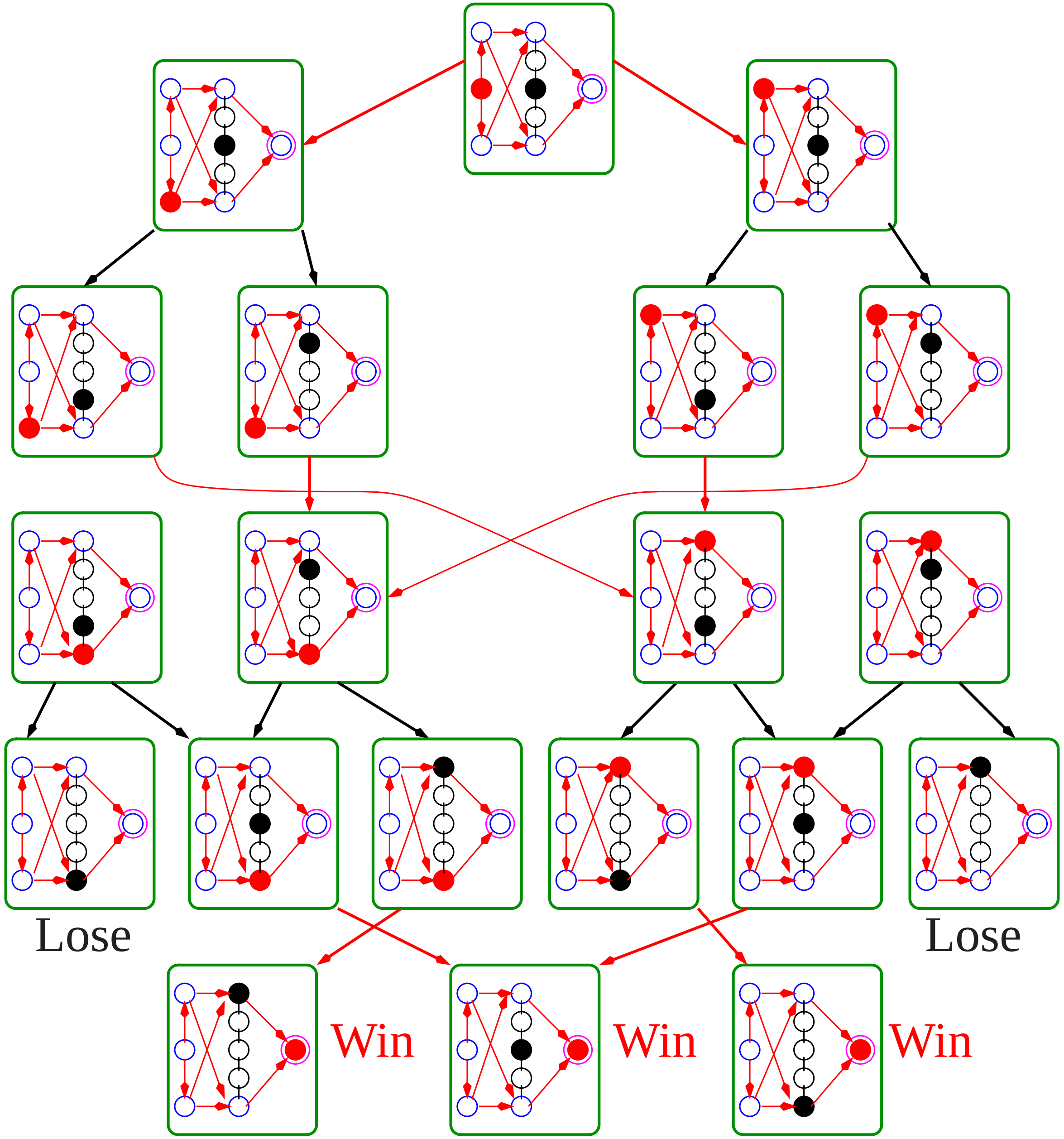
Computing $\mu R.\ \|win\| \cup Pre_{\exists}(R)$



Computing $\mu C. \|win\| \cup Pre_{\exists\forall}(C)$



A Good Strategy



Apply to Program Synthesis

The general approach considers a game $G = \langle \mathcal{G}, \varphi \rangle$ consisting of a state-transition diagram $\mathcal{G}$, whose transitions are partitioned into controllable and uncontrollable transitions, and a temporal formula φ, which the system should maintain.

In the previous examples, the formula was of the form $\diamondsuit$ *win*, requiring that a winning state is eventually reached. For such formulas, the set of winning states can be computed by the expression $\mu y.\, \mathit{win} \vee Pre_{\exists\forall}(y)$, and we can always obtain a memory-less strategy by removing some of the transitions.

Claim 1. *For every game $G = \langle \mathcal{G}, \varphi \rangle$ such that $\mathcal{G}$ is finite-state and φ is a propositional LTL formula, it is possible to compute the set of winning states by an appropriate fix-point expression.*

Furthermore, for the case that φ has one of the forms $\square\, p$, $\diamondsuit\, q$, or $\bigvee_{i=1}^{n}(\diamondsuit \square\, p_i \;\wedge\; \square \diamondsuit\, q_i)$ for state formulas p, q, p_i and q_i, then the game is winnable by red iff red has a winning memory-less strategy.

Different Solutions to Different Winning Conditions

When applied to controller synthesis, we denote the controlled predecessor by $\boxed{\Diamond} p$ with the meaning that $s \models \boxed{\Diamond} p$ 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.

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

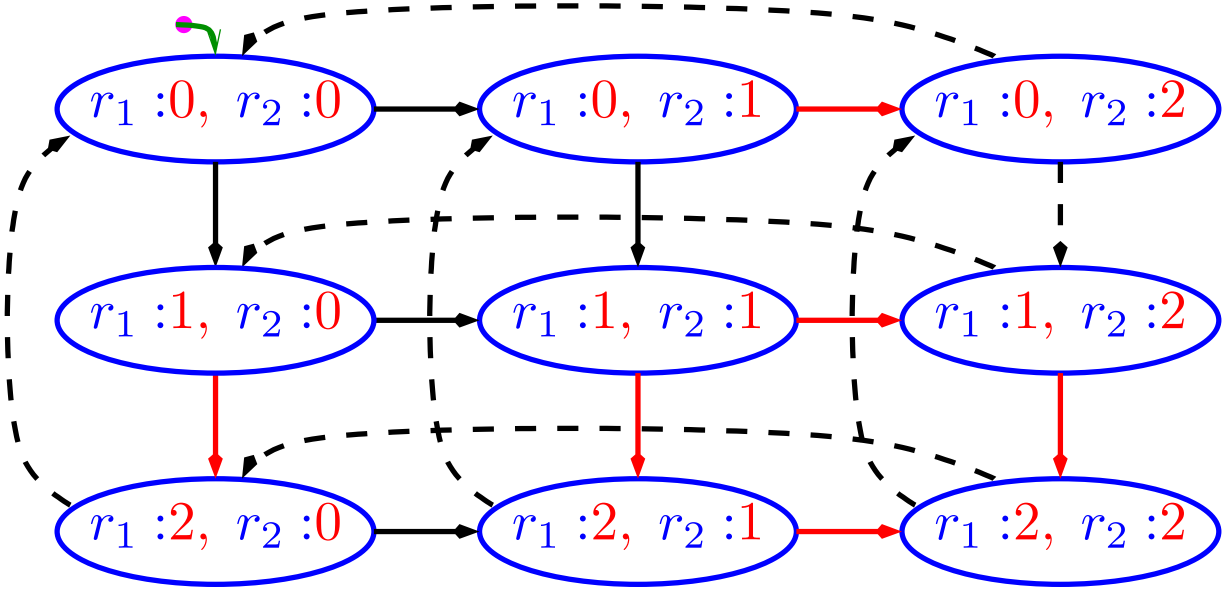
Winning Condition	Fixpoint Expression
$\Diamond\, W$	$\mu y. W \ \vee\ \boxed{\Diamond} y$
$\Box\, W$	$\nu y. W \ \wedge\ \boxed{\Diamond} y$
$\Box\, \Diamond\, W$	$\nu z \mu y. W \wedge \boxed{\Diamond} z \vee \boxed{\Diamond} y$

The last cases is based on the maximal fix-point soluion of the equation

$$z \quad = \quad \mu y. (W \wedge \boxed{\Diamond} z) \vee \boxed{\Diamond} y$$

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

Illustrate on MUX-SEM

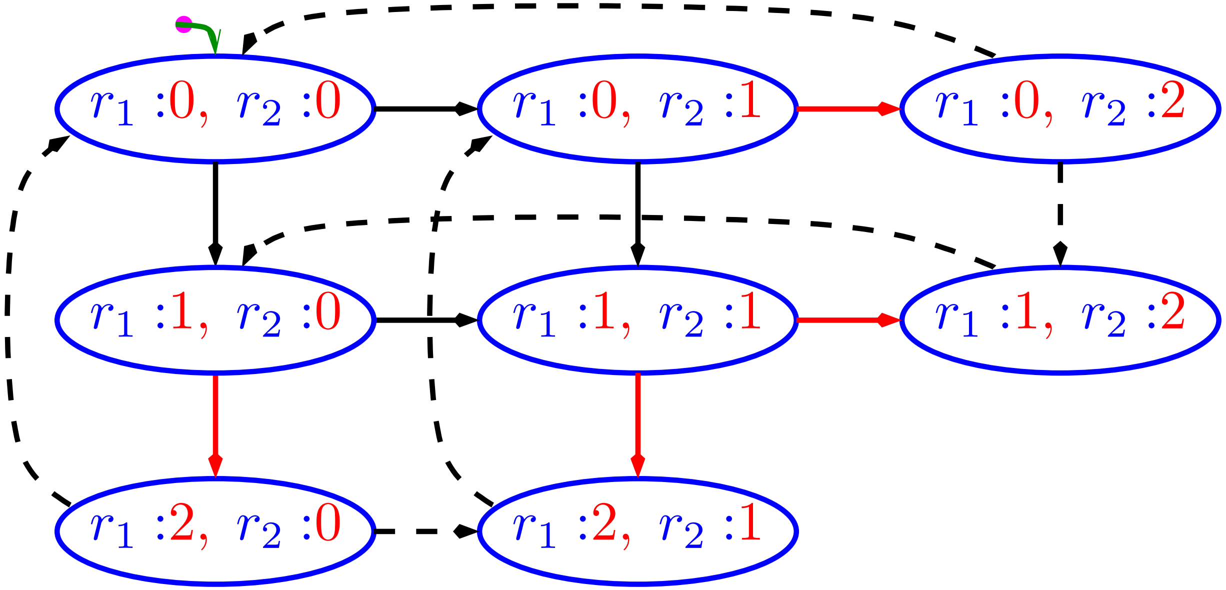


We wish to synthesize a program that guarantees

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

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

Applying the synthesis algorithm for this formula, we obtain



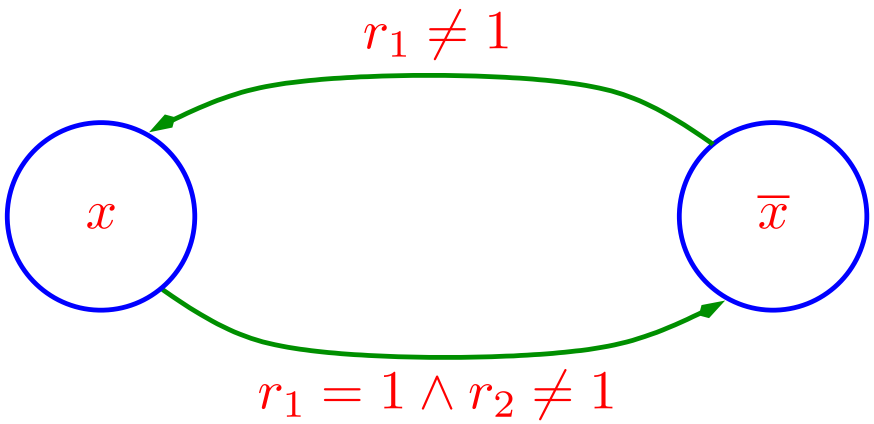
Have still to satisfy

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

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 $r_2 \neq 1$.



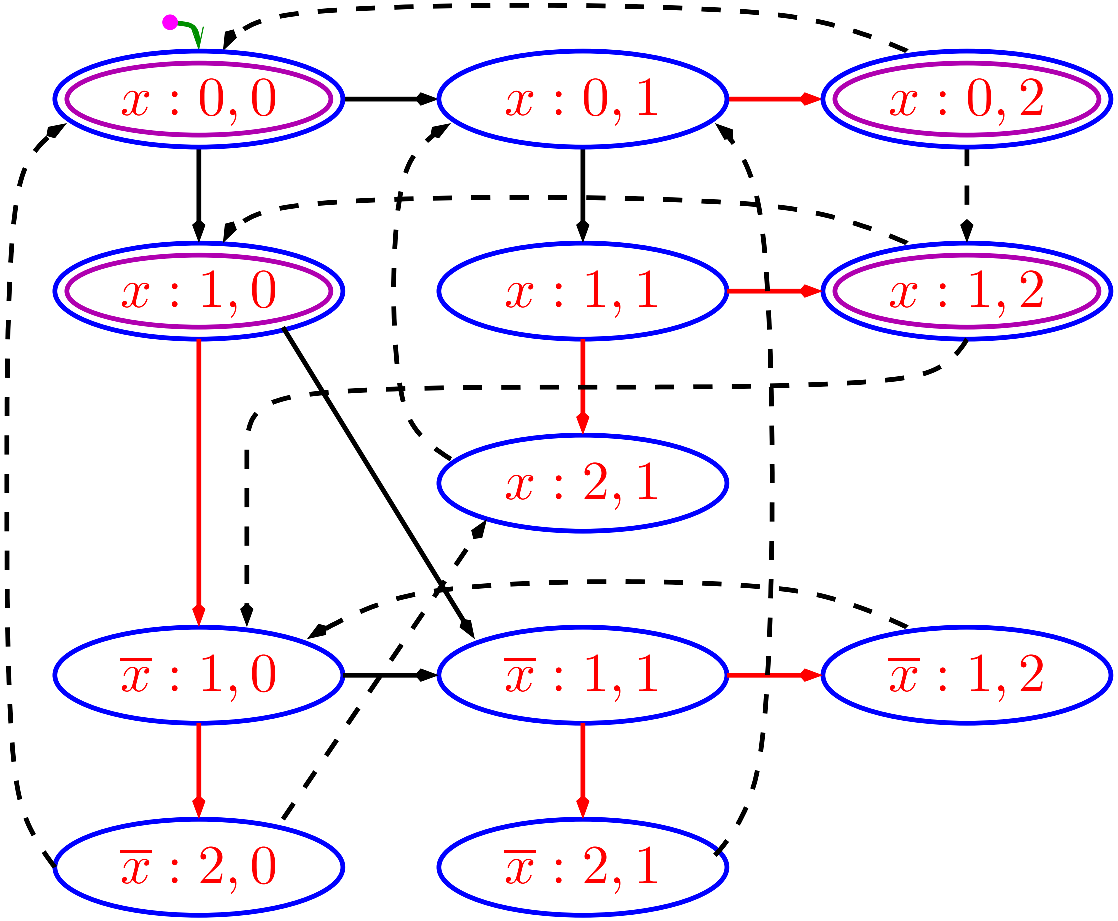
This automaton can be defined as an FDS A with the transition relation:

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

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

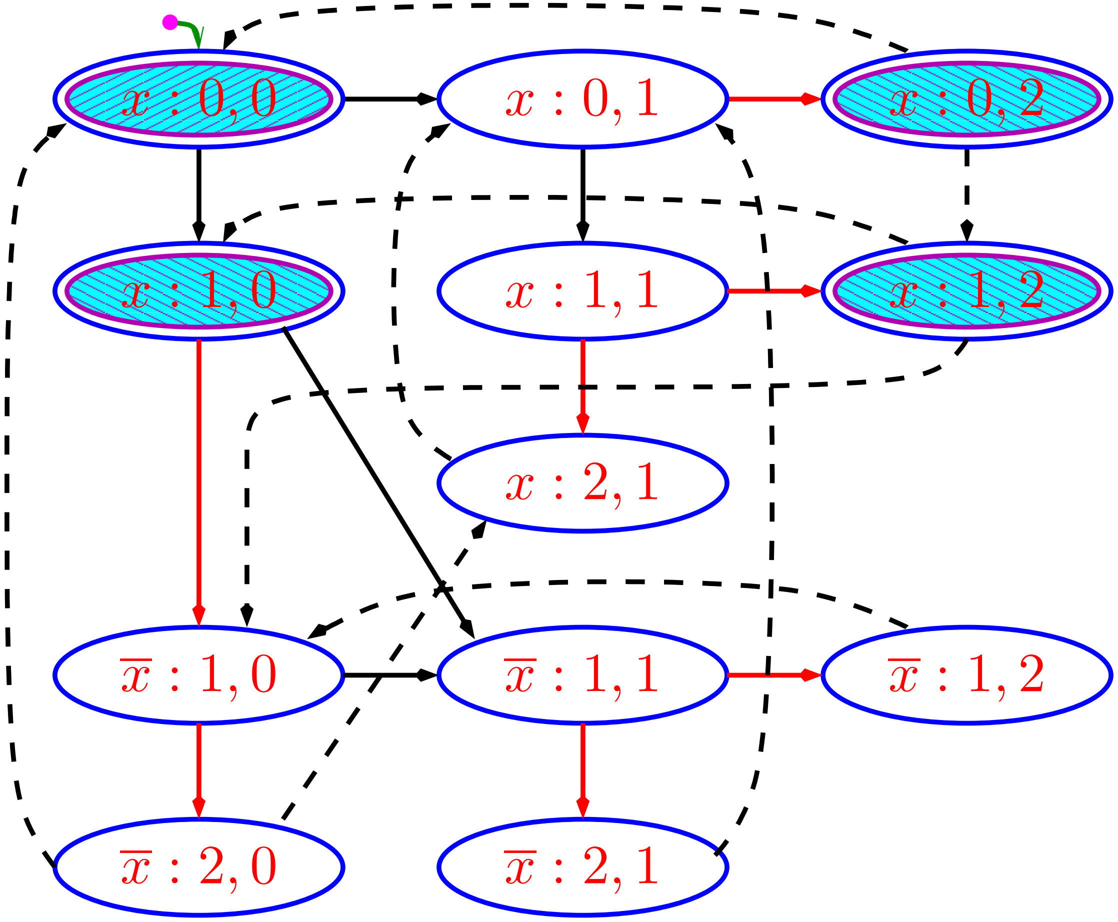
Form the Parallel Composition and Solve

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

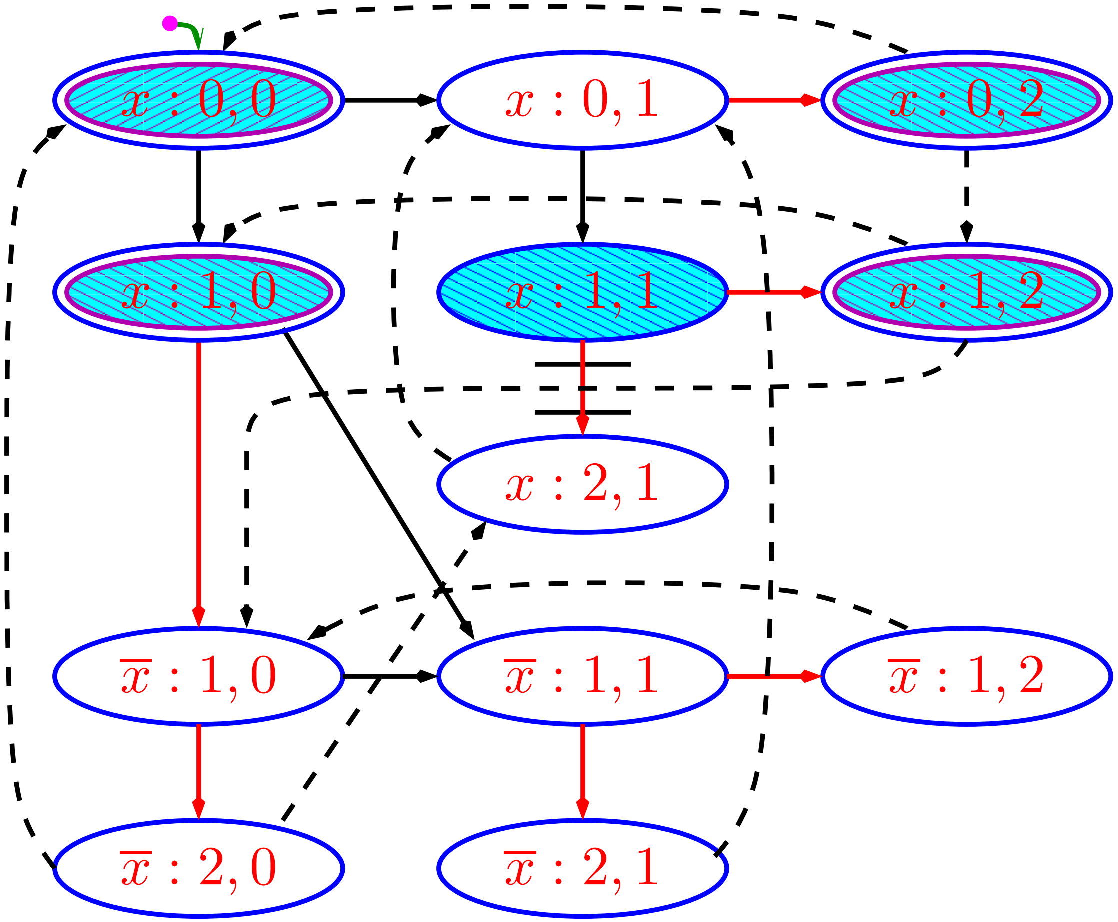


Solving: Step 0

Mark all immediately winning States as members of .

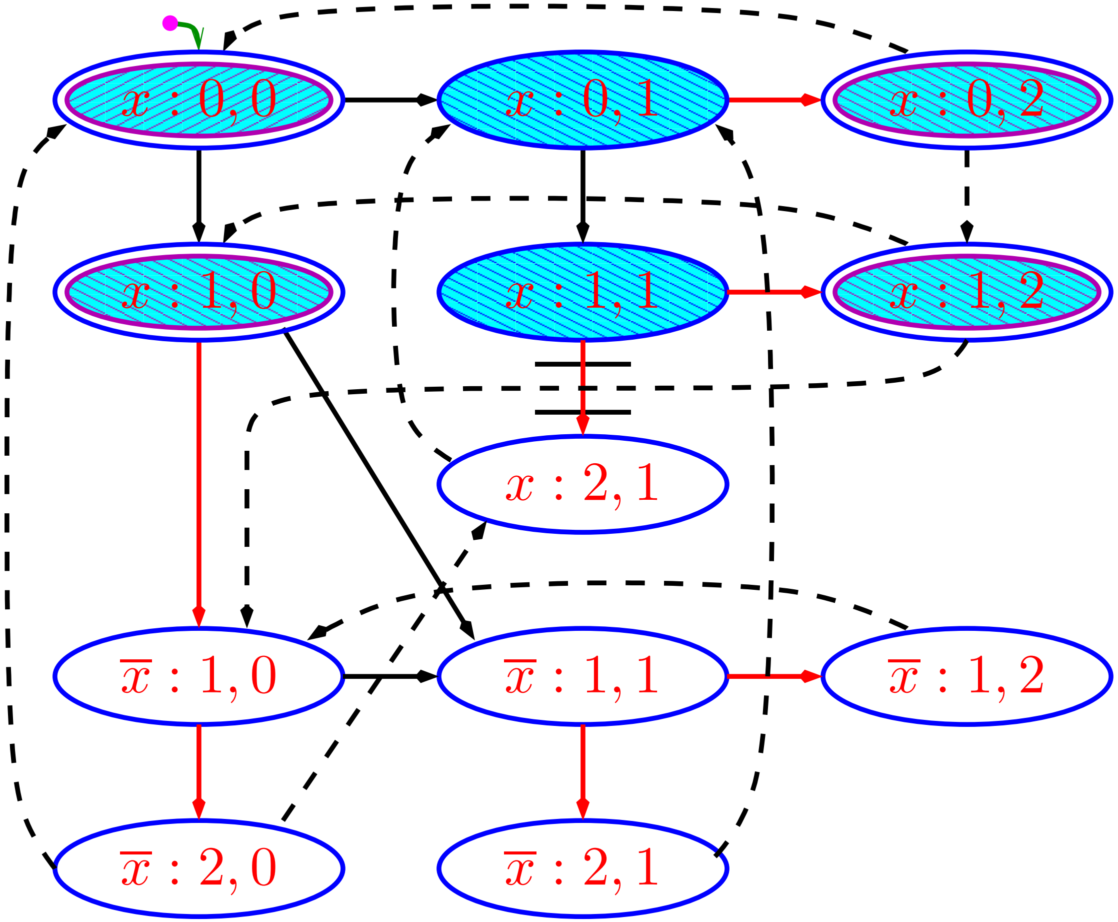


Solving: Step 1



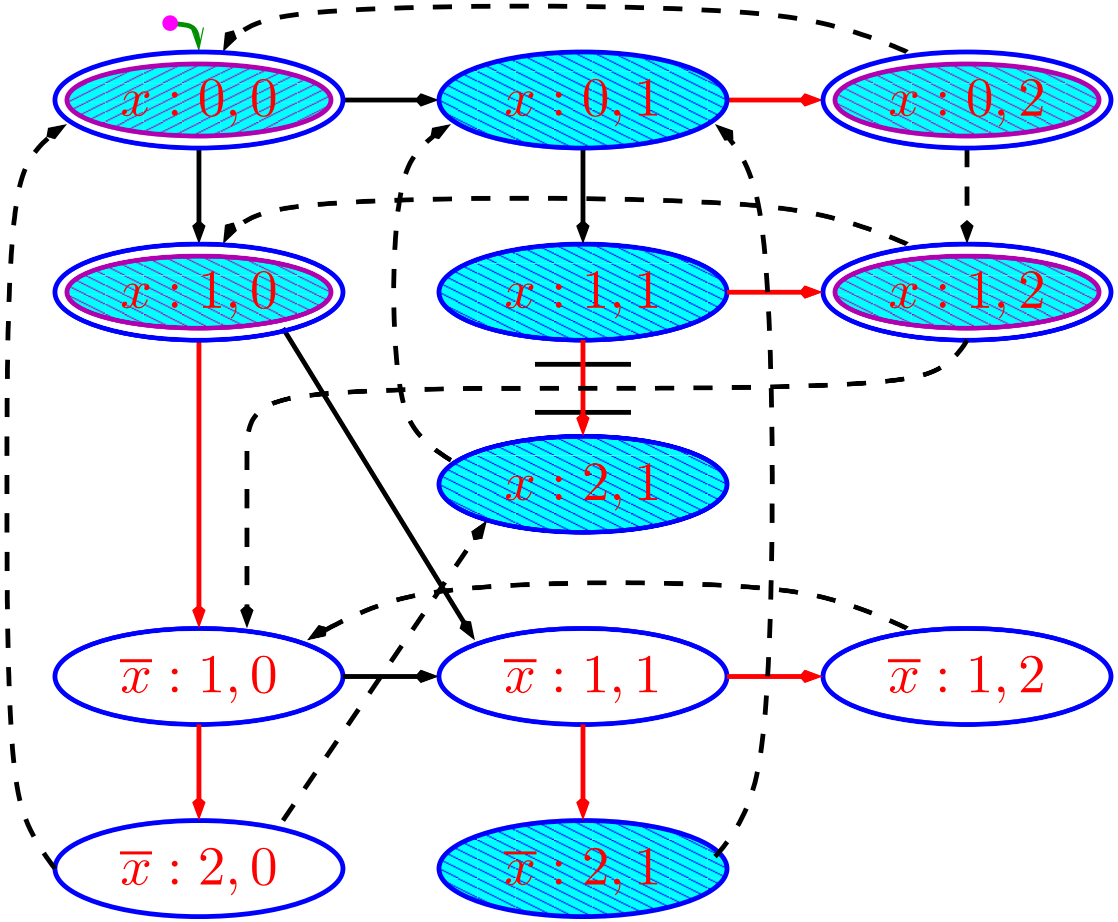
Add state $(x : 1, 1)$ since it has a winning successor.

Solving: Step 2



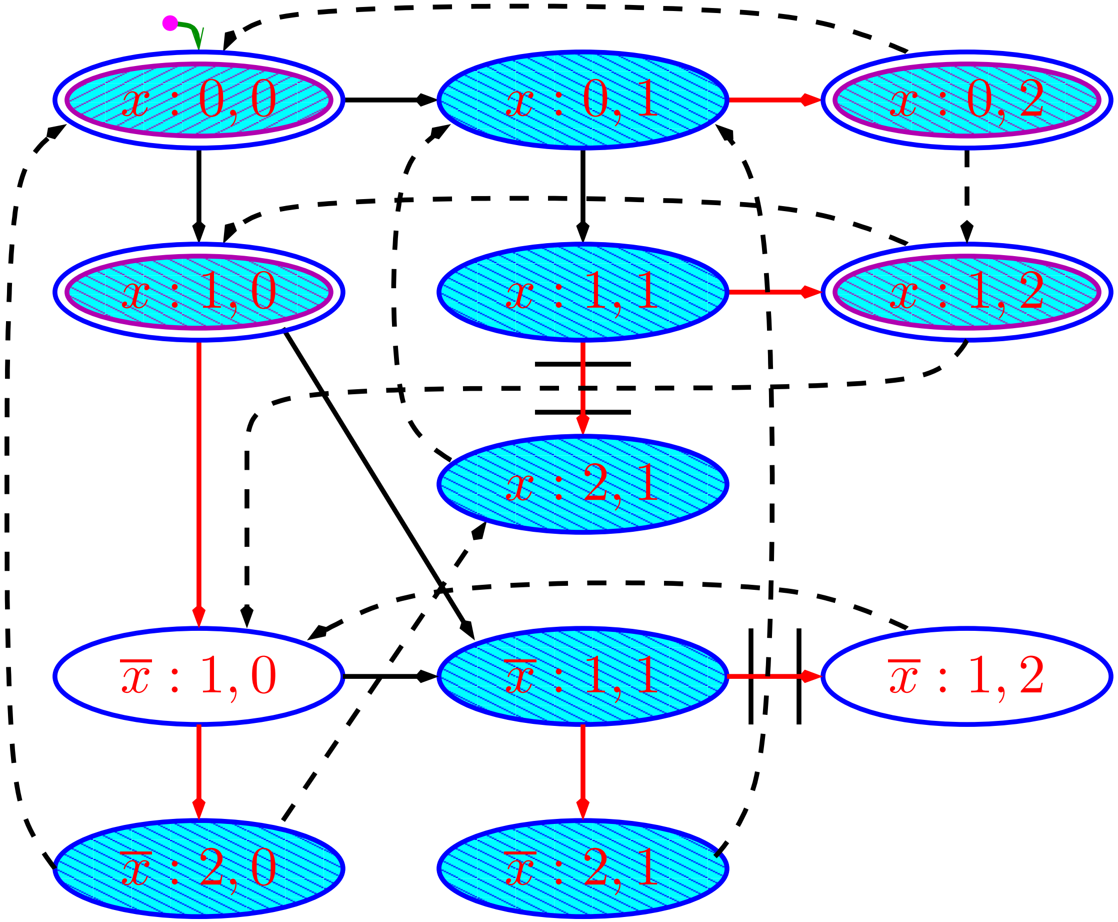
Add state $(x : 0, 1)$ since it has a winning successor.

Solving: Step 3



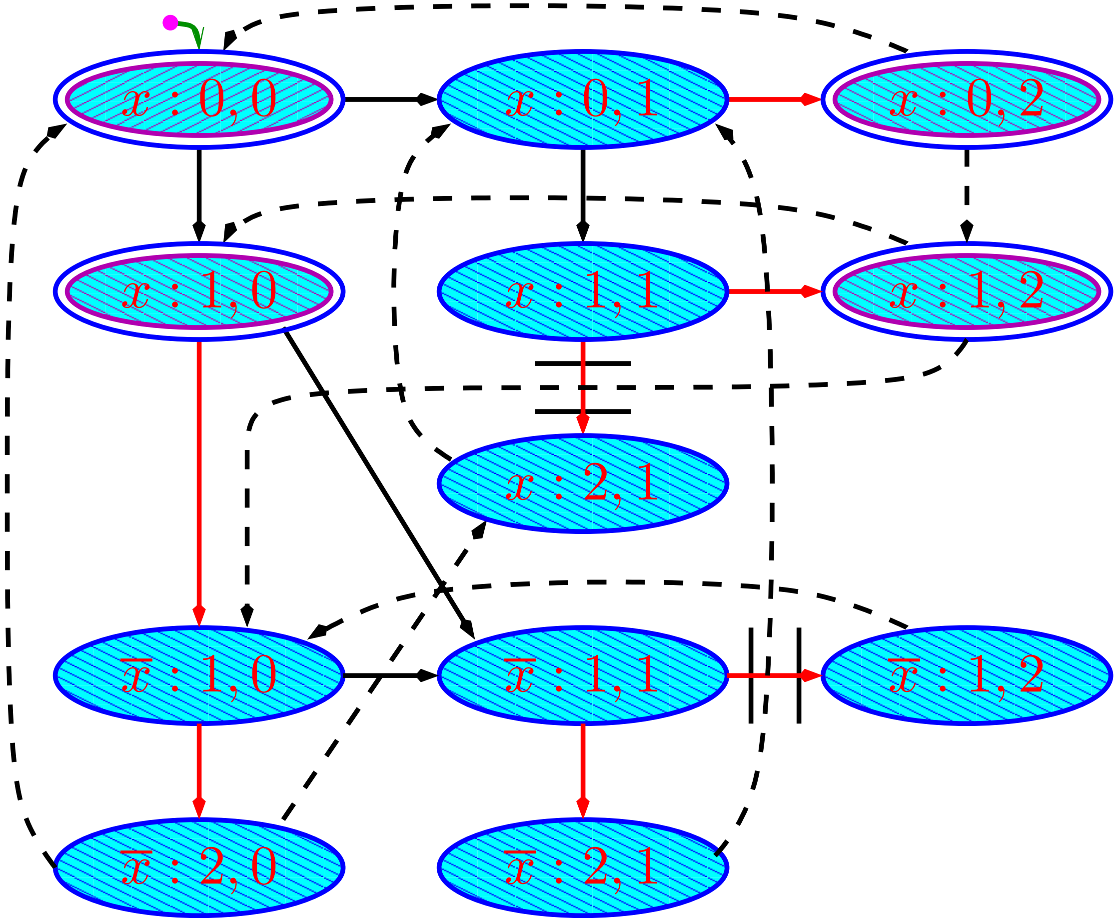
Add states $(x : 2, 1)$ and $(\overline{x} : 2, 1)$ since they each have only winning successors.

Solving: Step 4



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

Solving: Step 5



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

Note the ultimately periodic sequence:

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

Program Synthesis from LTL Specification

It is not always necessary to start with a given “plant”. We can synthesize direcly from LTL specifications.

Property-Based System Design

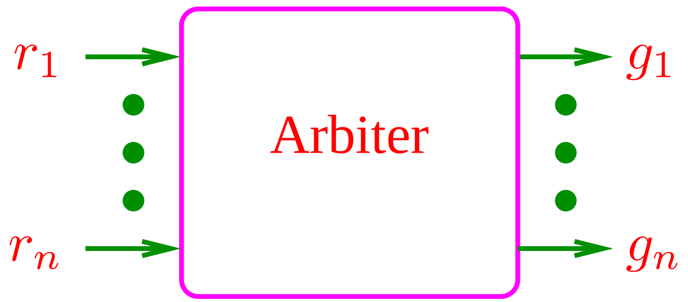
While the rest of the world seems to be moving in the direction of model-based design (see UML), we persisted with the vision of property-based approach.

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

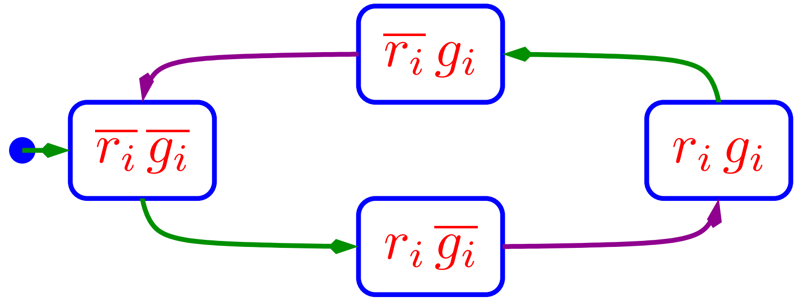
This is currently studied in the hardware-oriented European project PROSYD.

Example Specification

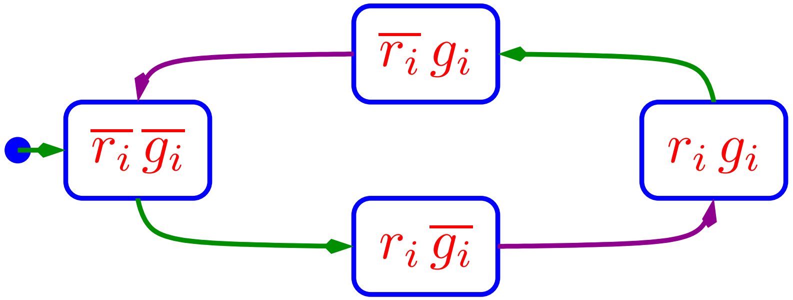
Consider a specification for an arbiter.



The protocol for each client:



The Specification



Assumptions (Constraints on the Environment)

$$A: \quad \bigwedge_i \left(\overline{r_i} \quad \wedge \quad (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \quad \wedge \quad r_i \wedge g_i \Rightarrow \diamondsuit\, \overline{r_i}\right)$$

Guarantees (Expectations from System)

$$G: \quad \bigwedge_{i \neq j} \square \neg(g_i \wedge g_j) \quad \wedge \quad \bigwedge_i \left(\overline{g_i} \quad \wedge \quad \begin{pmatrix} r_i = g_i & \Rightarrow & \bigcirc g_i = g_i & \wedge \\ r_i \wedge \overline{g_i} & \Rightarrow & \diamondsuit\, g_i & \wedge \\ \overline{r_i} \wedge g_i & \Rightarrow & \diamondsuit\, \overline{g_i} & \end{pmatrix}\right)$$

Total Specification

$$\varphi: \quad A \rightarrow G$$

Checking that a Specification is Feasible

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

Inconsistency

$$\diamondsuit g \quad \wedge \quad \square \neg g$$

Unrealizability For a system

Realizing the specification

$$g \quad \longleftrightarrow \quad \diamondsuit r$$

requires clairvoyance.

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.
- $\rho_2(X, Y, X', Y')$ — Transition relation for player 2.
- 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

$$\pi : \quad s_0, s_1, s_2, \ldots,$$

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 $\pi : s_0, s_1, \ldots$ is said to be compatible with strategy σ_1 if, for every $j \geq 0$, $s_{j+1}[X] = \sigma_1(s_0, \ldots, s_j)$.

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

$$r_i \ \wedge \ \neg g_i \quad \rightarrow \quad r_i'$$

- 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

$$
\begin{array}{l}
\bigwedge_i \left(\overline{r_i} \quad \wedge \quad (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \quad \wedge \quad r_i \wedge g_i \Rightarrow \Diamond\, \overline{r_i}\right) \quad \to \\
\qquad \bigwedge_{i \neq j} \Box\, \neg(g_i \wedge g_j) \quad \wedge \quad \bigwedge_i \left(\overline{g_i} \quad \wedge \quad \left[\begin{array}{lcll} r_i = g_i & \Rightarrow & \bigcirc g_i = g_i & \wedge \\ r_i \wedge \overline{g_i} & \Rightarrow & \Diamond\, g_i & \wedge \\ \overline{r_i} \wedge g_i & \Rightarrow & \Diamond\, \overline{g_i} & \end{array}\right]\right)
\end{array}
$$

We take the following game components:

$$
\begin{array}{rl}
X \cup Y : & \{r_i \mid i = 1, \ldots, n\} \cup \{g_i \mid i = 1, \ldots, n\} \\
\Theta : & \bigwedge_i (\overline{r_i} \;\wedge\; \overline{g_i}) \\
\rho_1 : & \bigwedge_i ((r_i \neq g_i) \to (r_i' = r_i)) \\
\rho_2 : & \bigwedge_{i \neq j} \neg(g_i' \;\wedge\; g_j') \quad \wedge \quad \bigwedge_i ((r_i = g_i) \to (g_i' = g_i)) \\
\varphi : & \bigwedge_i (r_i \wedge g_i \Rightarrow \Diamond\, \overline{r_i}) \quad \to \quad \bigwedge_i ((r_i \wedge \overline{g_i} \Rightarrow \Diamond\, g_i) \;\wedge\; (\overline{r_i} \wedge g_i \Rightarrow \Diamond\, \overline{g_i}))
\end{array}
$$

Solving Games for 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

$$\Diamond\Box\, p_1 \;\vee\; \Diamond\Box\, p_2 \;\vee\; \cdots \;\vee\; \Diamond\Box\, p_m \vee\; \Box\Diamond q_1 \wedge\; \Box\Diamond q_2 \;\wedge\; \cdots \;\wedge\; \Box\Diamond q_n$$

equivalently,

$$(\Box\Diamond p_1 \wedge\; \Box\Diamond p_2 \;\wedge\; \cdots \;\wedge\; \Box\Diamond p_m) \quad\rightarrow\quad \Box\Diamond q_1 \wedge\; \Box\Diamond q_2 \;\wedge\; \cdots \;\wedge\; \Box\Diamond q_n$$

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

$$(\cdots \;\wedge\; g_i \Rightarrow \Diamond \neg r_i \;\wedge\; \cdots) \quad\rightarrow\quad \cdots \;\wedge\; r_i \Rightarrow \Diamond\, g_i \;\wedge\; \cdots$$

Response vs. Recurrence Properties

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

$$p \Rightarrow \Diamond q \qquad \sim \qquad \Box\Diamond((\neg p)\ \mathcal{B}\ q)$$

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

Rewrite	$r_i \wedge g_i \Rightarrow \Diamond \overline{r_i}$	as	$\Box \Diamond \neg(r_i \wedge g_i)$
Rewrite	$r_i \wedge \overline{g_i} \Rightarrow \Diamond g_i$ and $\overline{r_i} \wedge g_i \Rightarrow \Diamond \overline{g_i}$	as	$\Box \Diamond (r_i = g_i)$

The Solution

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

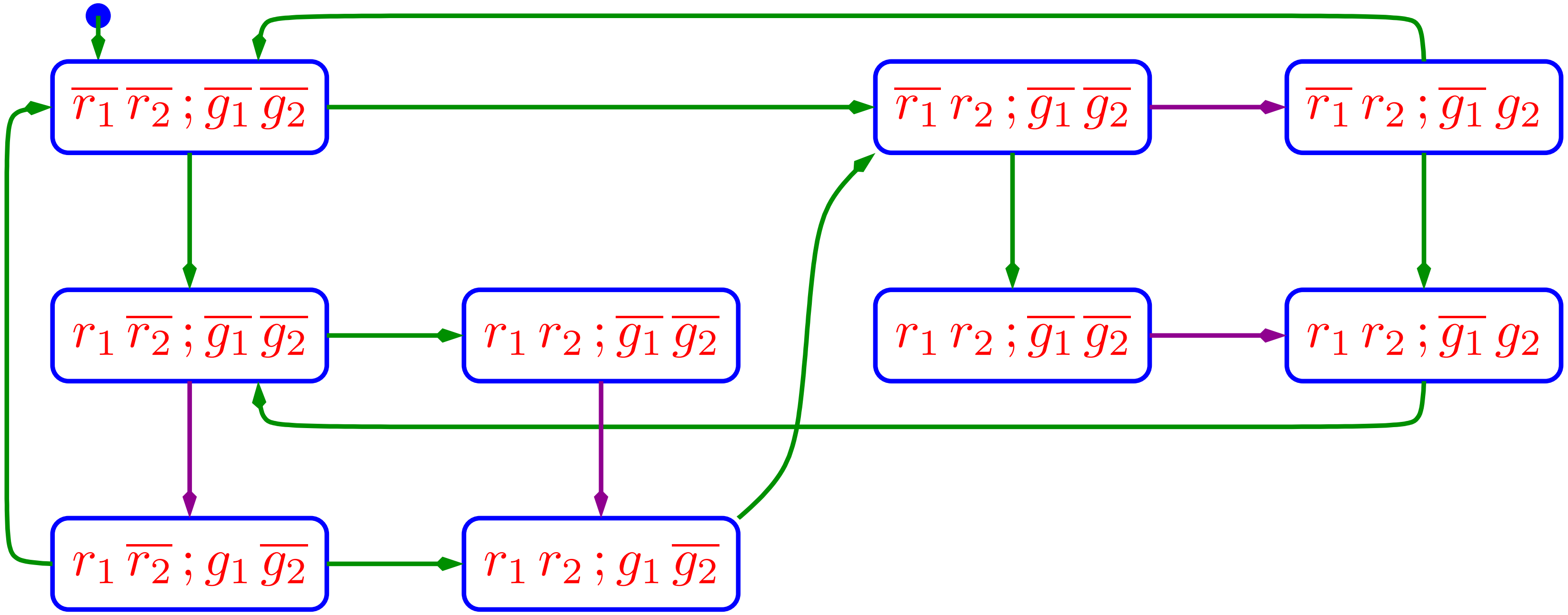
$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{j=1}^{m} \nu X (q_1 \wedge \boxed{\diamond} Z_2 \ \vee \ \boxed{\diamond} Y \ \vee \ \neg p_j \wedge \boxed{\diamond} X) \right) \\ \mu Y \left(\bigvee_{j=1}^{m} \nu X (q_2 \wedge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

where

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

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 Arbiter(N)

N	Recurrence Properties	Design Size	Response Properties
4	0.05	181	0.33
6	0.06	645	0.89
8	0.13	1147	1.77
10	0.25	1793	3.04
12	0.48	2574	4.92
14	0.87	3499	7.30
16	1.16	4559	10.57
18	1.51	5767	15.05
20	1.89	7108	20.70
25	3.03	11076	43.69
30	4.64	15925	88.19
35	6.78	21647	170.50
40	9.50	28238	317.33

Extent of Properties Class

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

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

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

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

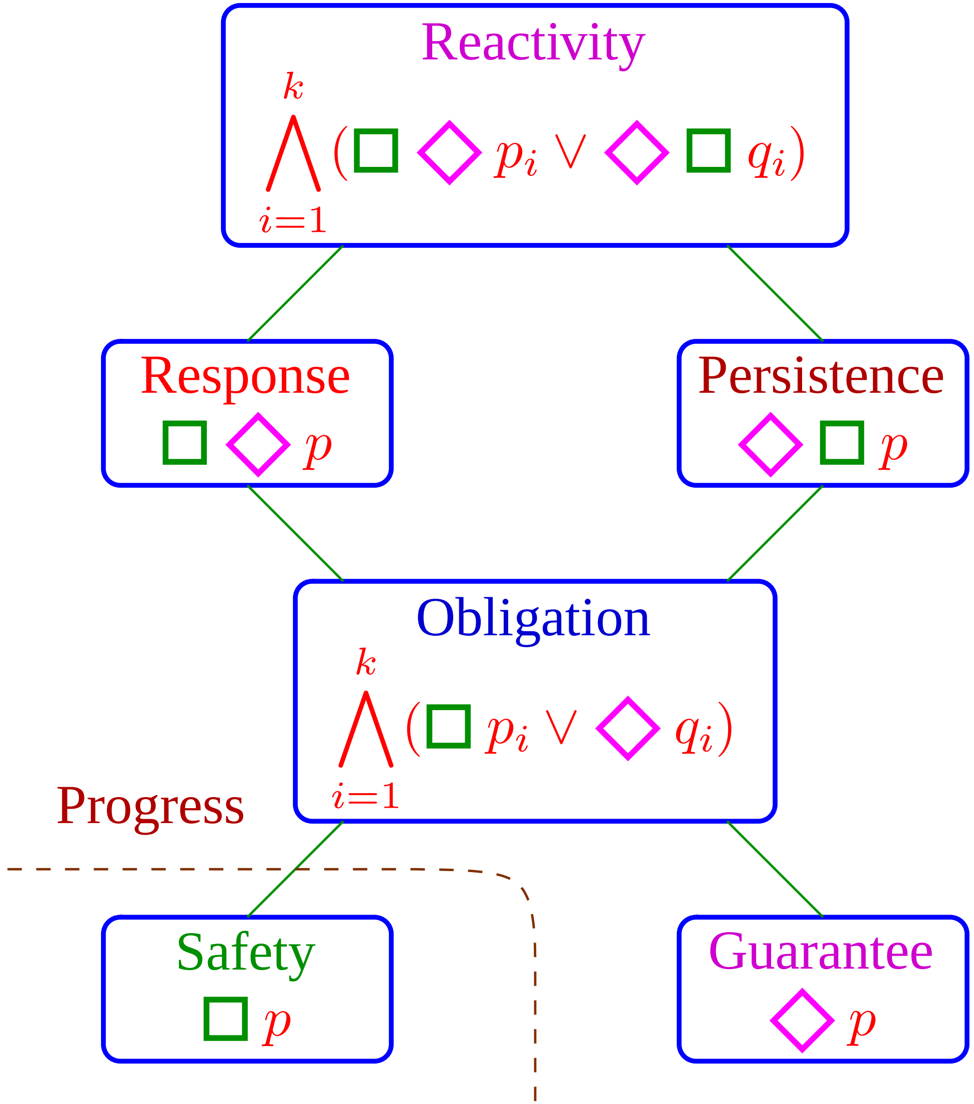
$$x' = (q \quad \vee \quad x \wedge \neg p)$$

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

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 representing response properties as recurrence.

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.