Synthesis of Designs from Temporal Specifications

Nir Piterman[1], Amir Pnueli[2], Yaniv Sa'ar[3]

1) EPFL, Lausanne
2) New York University and Weizmann Institute of Sciences
3) Ben Gurion University, Beer Sheva

TCC'06, March, 2006

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, e.g. an LTL formula.

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

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

Example of a Specification

Behavioral Specification:

$$\square\,(\bigcirc x = (y \oplus \bigcirc y))$$

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.

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) environment.

In the same work he has 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 $\Box p$, the controller can be synthesized in linear time.

In 1995, 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.

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.

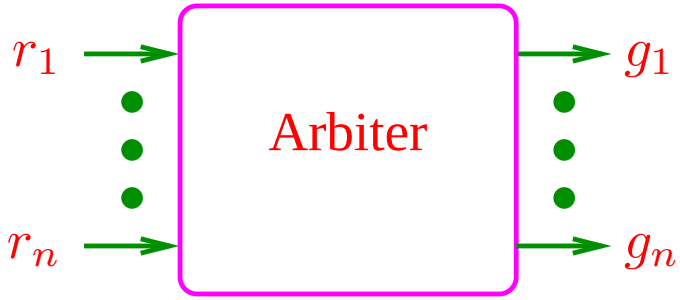
This is currently studied in the hardware-oriented European project PROSYD.

Design synthesis is needed in two places in the development flow:

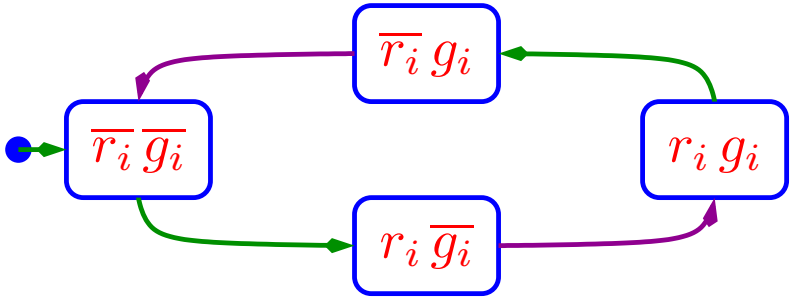
- Automatic synthesis of small blocks whose time and space efficiency are not critical.
- As part of the specification analysis phase, ascertaining that the specification is realizable.

Example Specification

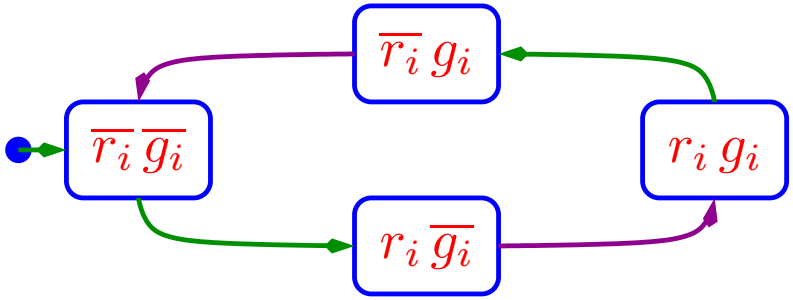
Reconsider a specification for an arbiter.



The protocol for each client:



The Specification



Assumptions (Constraints on the Environment)

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

Guarantees (Expectations from System)

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

Total Specification

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

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

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

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

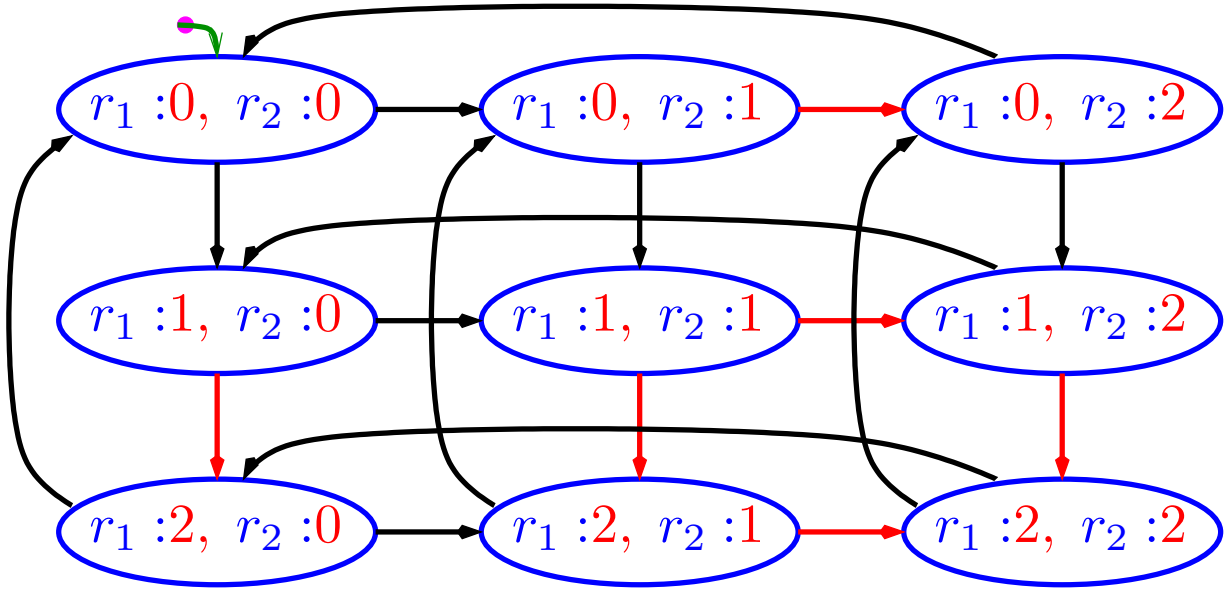
$$\bigwedge_i \Big(\overline{r_i} \;\wedge\; (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \;\wedge\; r_i \wedge g_i \Rightarrow \Diamond\, \overline{r_i}\Big) \;\rightarrow$$
$$\bigwedge_{i\neq j} \Box\, \neg(g_i \wedge g_j) \;\wedge\; \bigwedge_i \left(\overline{g_i} \;\wedge\; \left(\begin{array}{lll} r_i = g_i & \Rightarrow & \bigcirc g_i = g_i \;\wedge \\ r_i \wedge \overline{g_i} & \Rightarrow & \Diamond\, g_i \;\wedge \\ \overline{r_i} \wedge g_i & \Rightarrow & \Diamond\, \overline{g_i} \end{array}\right)\right)$$

We take the following game components:

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

In Controller Synthesis

We are given a system (plant), such as



And asked to synthesize a controller that guarantees

$$\Box\, \neg(r_1 = 2 \ \wedge\ r_2 = 2) \ \wedge\ (\Box\, \Diamond\, (r_1 \neq 1) \ \wedge\ \Box\, \Diamond\, (r_2 \neq 1))$$

In Design Synthesis, the plant is derived from the safety part of the LTL specification.

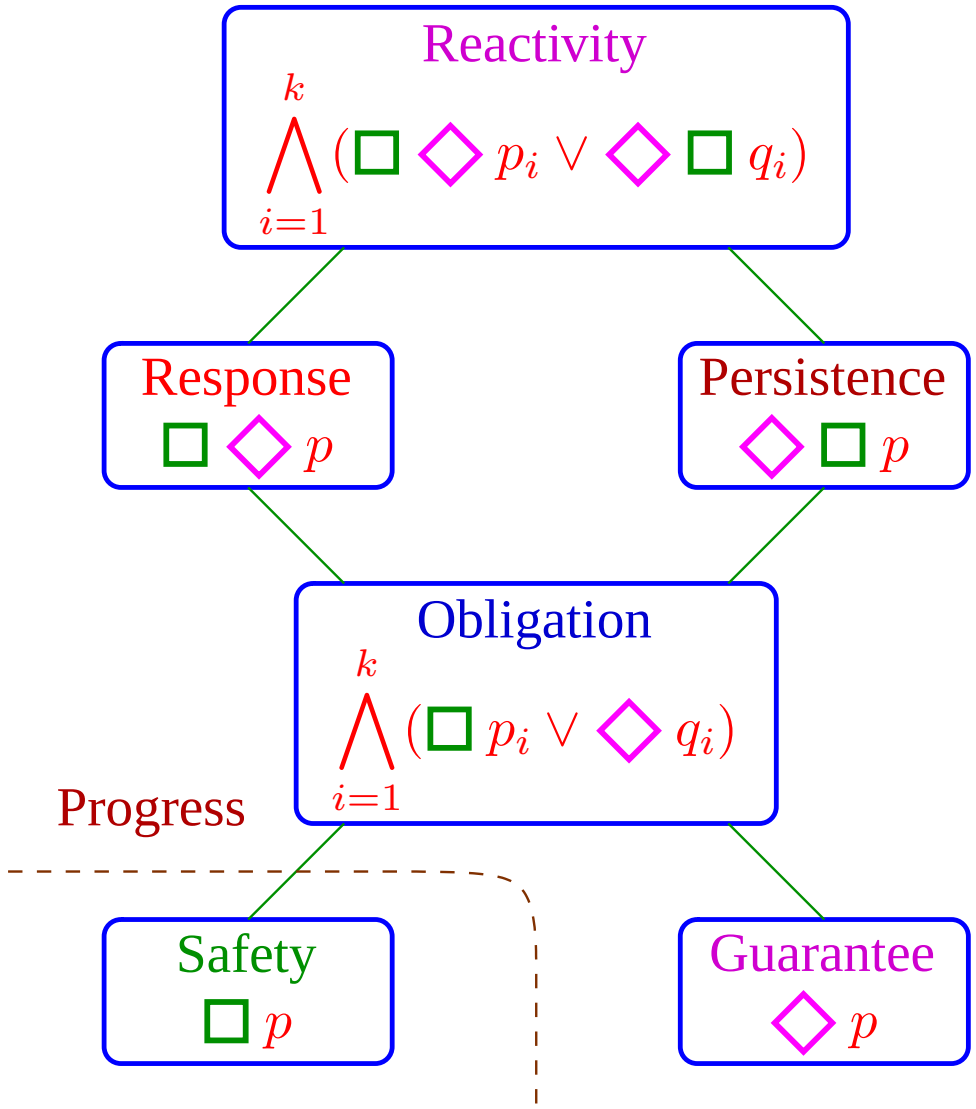
Solving the Game

The double exponential complexity is a direct consequence of the automatic translation of LTL into Büchi automata.

One of the messages of this paper is:

> Do not be too hasty to tranlsate LTL into automata. Try first to locate the formula within the temporal hierarchy.

Hierarchy of the Temporal Properties



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

Different Solutions to Different Winning Conditions

When applied to game solving, we denote the controlled predecessor by $\boxdot p$ with the meaning that $s \models \boxdot p$ 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:

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

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
$\Diamond W$	$\mu y.\, W \vee \boxdot y = W \vee \boxdot W \vee \boxdot\boxdot W \vee \cdots$
$\Box W$	$\nu y.\, W \wedge \boxdot y = W \wedge \boxdot W \wedge \boxdot\boxdot W \wedge \cdots$
$\Box\Diamond W$	$\nu z \mu y.\, W \wedge \boxdot z \vee \boxdot y$

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

$$z \quad = \quad \mu y.\, (W \wedge \boxdot z) \vee \boxdot y$$

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

An Iterative View of the Game Solving Algorithms

Solving the game for $\Diamond q$ — $\mu Y.\left(q \vee \Box Y\right)$.

$Y := 0;$ **Fix** (Y) $[Y := q \vee \Box Y]$

Solving the game for $\Box p$ — $\nu X.\left(p \wedge \Box X\right)$.

$X := 1$
Fix (X)
$[X := p \wedge \Box X]$

Solving the game for $\Box \Diamond q$ — $\nu Z \mu Y.\left((q \wedge \Box z) \vee \Box Y\right)$.

$Z := 1$
Fix (Z)
$$\left[\begin{array}{l} G := q \wedge \Box Z \\ Y := 0 \\ \textbf{Fix}\ (Y) \\ \quad [Y := G \vee \Box Y] \\ Z := Y \end{array}\right]$$

Simple Reactivity[1] Games

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

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

to

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

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

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

equivalently,

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

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

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

The Solution

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

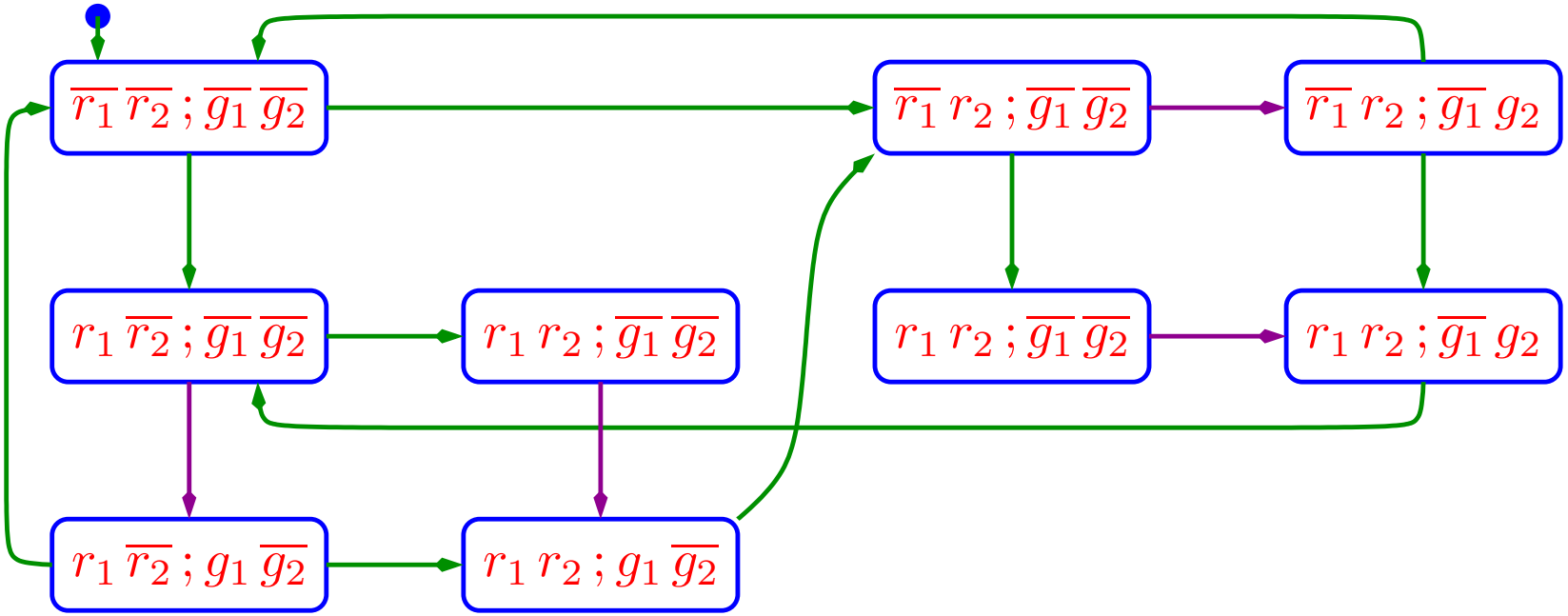
$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{j=1}^{m} \nu X (q_1 \wedge \Box Z_2 \vee \Box Y \vee \neg p_j \wedge \Box X) \right) \\ \mu Y \left(\bigvee_{j=1}^{m} \nu X (q_2 \wedge \Box Z_3 \vee \Box Y \vee \neg p_j \wedge \Box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

where

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

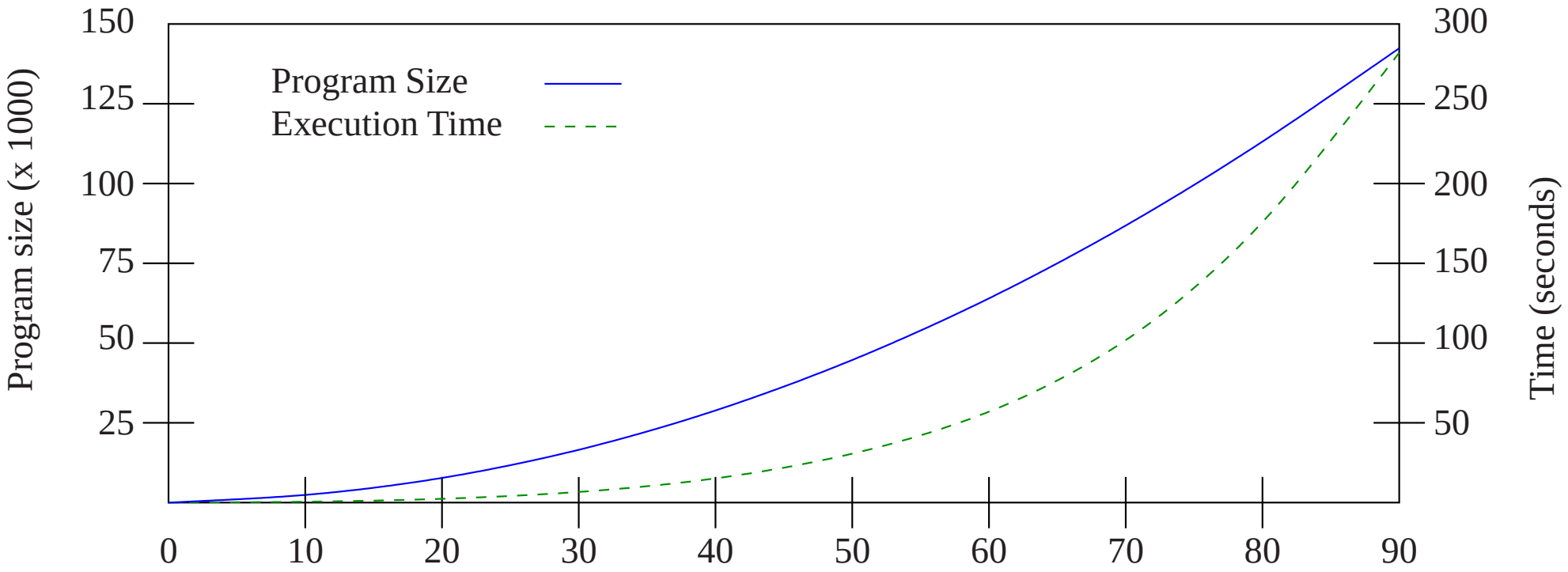
Results of Synthesis

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)



Conclusions

- It is possible to perform design synthesis for restricted fragments of LTL in acceptable time.
- The tractable fragment (React(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 $N^{(2k+1)}$.