Lectures Outline

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

Classical (Continuous Time) Control

Environment \( \rightarrow \) Plant \( \rightarrow \) Controller

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 R tries to reach the goal. The blocker B tries to intercept and stop the runner.
State Transitions Diagram
Is the Goal Reachable?

All of our algorithms will be computing sets of states out of the state-transition diagram. Let \( \| \text{win} \| \) denote the set of states labeled by the \textit{win} proposition. Let \( \rho \) 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 \( \text{Pre}_\exists \) which computes the set of all one-step predecessors of the states in \( S \). That is,

\[
\text{Pre}_\exists(S) = \{ s \mid s \text{ has a } \rho\text{-successor in } S \}
\]

Recursively, we define a state \( s \) to be goal reaching if either \( s \in \| \text{win} \| \) or \( s \) has a goal reaching successor. That is,

\[
R = \| \text{win} \| \cup \text{Pre}_\exists(R)
\]

We may expect that the solution to this fix-point equation, will give us the set of all states from which \( \| \text{win} \| \) is reachable.
Problem: Not Every Solution is Satisfactory

Consider the diagram:

Both $R_{0,1} = \{s_0, s_1\}$ and $R_{0,2} = \{s_0, s_1, s_2\}$ satisfy the equation
\[ R = \|\text{win}\| \cup \text{Pre}_\exists(R) \]
but only $R_{0,1} = \{s_0, s_1\}$ captures the set of states from which $\|\text{win}\|$ is reachable.

**Conclusion:** We should take the minimal solution of the fix-point equation $R = \|\text{win}\| \cup \text{Pre}_\exists(R)$ which we denote by $\mu R. (\|\text{win}\| \cup \text{Pre}_\exists R)$

This minimal solution can be effectively computed by the iteration sequence:
\[
\begin{align*}
R_0 &= \emptyset \\
R_1 &= \|\text{win}\| \\
R_2 &= R_0 \cup \text{Pre}_\exists R_0 \\
R_3 &= R_1 \cup \text{Pre}_\exists R_1 \\
\cdots
\end{align*}
\]

Consequently, the goal is reachable from an initial state $s_0$ iff $s_0 \in \mu R. (\|\text{win}\| \cup \text{Pre}_\exists R)$. 

Verification and Synthesis of Reactive Programs, Rome, June, 2006
Computing $\mu R. \parallel \text{win} \parallel \cup Pre_3(R)$
Distinguishing between the two players, we define

\[
\begin{align*}
\text{Pre}_\exists(S) & = \{ s \mid \text{Some red successor of } s \text{ is in } C \} \\
\text{Pre}_\forall(S) & = \{ s \mid \text{All black successors of } s \text{ are in } C \}
\end{align*}
\]

The two operators can be combined, and the expression

\[
\text{Pre}_C(C) = \text{Pre}_\exists(\text{Pre}_\forall(C))
\]

denotes the set of states \( s \) which have at least one \textbf{red} successor \( s_1 \) all of whose \textbf{black} successors belong to \( C \). If we think about the moves as taken in turn by two players, then \( \text{Pre}_C(C) \) denotes the states from which the \textbf{red} player can force the game after a complete round (each player making one move) into a \( C \)-state.

The expression \( \text{control}(\text{win}) = \mu C. \text{||win||} \cup \text{Pre}_C(C) \) characterizes all the states from which the \textbf{red} player can force a visit to a \textbf{win} state in a finite number of moves.
Computing $\mu C. \parallel \text{win} \parallel \cup Pre_c(C)$
Conclusions

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

However, $R$ cannot force a win.
The Modified Runner Blocker System

Additional transitions have been added to the runner.
Game Tree for the Modified System
Computing $\mu R. \parallel \text{win} \parallel \cup \text{Pre}_3(R)$
Computing $\mu C. \| \text{win} \| \cup \text{Pre}_C(C)$
A Winning Strategy
Let us apply the controller synthesis paradigm to synthesis of programs (or designs, in general).
Example Design: Arbiter

Consider a specification for an arbiter.

The protocol for each client:

Required to satisfy

$$\bigwedge_{i \neq j} \Box \neg (g_i \land g_j) \land \bigwedge_i \Box \Diamond (g_i = r_i)$$
Start by **Controller Synthesis**

Assume a given **platform** (plant), identifying **controllable** (system) and **uncontrollable** (environment) transitions:

By default every node is connected to itself by both green and red transitions. A complete move consists of a red edge followed by a green edge, visiting at most two different states. Also given is an LTL specification (**winning condition**):

$$\varphi: \square \neg (g_1 \land g_2) \land \square \Diamond (g_1 = r_1) \land \square \Diamond (g_2 = r_2)$$
A game is given by $G : \langle V = X \cup Y, \Theta, \rho_1, \rho_2, \varphi \rangle$, 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 $\Sigma$ denote the set of all states.

- $\Theta$ — 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).

- $\varphi$ — The winning condition. An LTL formula characterizing the plays which are winning for player 2.

A state $s_2$ is said to be a $G$-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 $G : \langle V, \Theta, \rho_1, \rho_2, \varphi \rangle$ be a game. A play of $G$ is an infinite sequence of states

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

satisfying:

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

A play $\pi$ 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 $\sigma_1$ if, for every $j \geq 0$, $s_{j+1}[X] = \sigma_1(s_0, \ldots, s_j)$.

Strategy $\sigma_1$ is winning for player 1 from state $s$ if all $s$-originated plays compatible with $\sigma_1$ 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 $G$ is said to be winning for player 2 (player 1, respectively) if all (some) initial states are winning for 2 (1, respectively).

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.

When applying controller synthesis, the platform provides the transition relations $\rho_1$ and $\rho_2$, as well as the initial condition.

Thus, the essence of synthesis under the controller framework is an algorithm for computing the set of winning states for a given platform and specification $\varphi$. 
The Controlled Predecessor

As in symbolic model checking, computing the winning states involves fix-point computations over a basic predecessor operator. For model checking the operator is $\mathbf{E}\bigcirc p$ satisfied by all states which have a $p$-state as a successor.

For synthesis, we use the controlled predecessor operator $\blacklozenge p$. Its semantics can be defined by

$$\blacklozenge p : \forall X' : \rho_1(V, X') \rightarrow \exists Y' : \rho_2(V, V') \land p(V')$$

where $\rho_1$ and $\rho_2$ are the transition relations of the environment and system, respectively.

In our graphic notation, $s \models \blacklozenge p$ iff $s$ has at least one green $p$-successor, and all red successors different from $s$ satisfy $p$. 
Solving □ \( p \) Games, Iteration 0

The set of winning states for a specification □ \( p \) can be computed by the fix-point expression:

\[
\nu Y. \ p \land \lozenge Y = \ p \land \lozenge p \land \lozenge \lozenge p \land \cdots
\]

We illustrate this on the specification □ \( \neg (g_1 \land g_2) \).
Solving □ p Games, Iteration 1

The set of winning states for a specification □ p can be computed by the fix-point expression:

\[ \nu Y. p \land \Box Y = p \land \Box p \land \Box \Box p \land \cdots \]

We illustrate this on the specification □ \( \neg(g_1 \land g_2) \).

Iteration 1, \( Y_1 : \neg(g_1 \land g_2) \land \Box 1 \)
Solving ♦ \( q \) Games, Iteration 1

The set of winning states for a specification ♦ \( q \) can be computed by the fix-point expression:

\[
\mu Y. q \lor \Box Y = q \lor \Box q \lor \Box \Box q \lor \cdots
\]

We illustrate this on the specification ♦ \( (g_1 = r_1) \).

\[ Y_1 : (g_1 = r_1) \]
The set of winning states for a specification $\Diamond q$ can be computed by the fix-point expression:

$$\mu Y. q \lor \Box Y = q \lor \Box q \lor \Box \Box q \lor \cdots$$

We illustrate this on the specification $\Diamond (g_1 = r_1)$.
Solving $\Diamond q$ Games, Iteration 3

The set of winning states for a specification $\Diamond q$ can be computed by the fix-point expression:

$$\mu Y. q \lor \square Y = q \lor \square q \lor \square \square q \lor \cdots$$

We illustrate this on the specification $\Diamond (g_1 = r_1)$.

Iteration 3,

$$Y_3 : Y_2 \lor \square Y_2$$
Solving $\Diamond q$ Games, Iteration 4 (Final)

The set of winning states for a specification $\Diamond q$ can be computed by the fix-point expression:

$$\mu Y. q \lor \Box Y = q \lor \Box q \lor \Box \Box q \lor \cdots$$

We illustrate this on the specification $\Diamond (g_1 = r_1)$.

Iteration 4, $Y_4 : Y_3 \lor \Box Y_3$
A game for a winning condition of the form \( \square \Diamond q \) can be solved by the fix-point expression:

\[
\nu Z \mu Y. q \land \Box Z \lor \Box Y
\]

This is based on the maximal fix-point solution of the equation

\[
Z = \mu Y. (q \land \Box Z) \lor \Box Y
\]

This nested fix-point computation can be computed iteratively by the program:

\[
\begin{align*}
Z &:= 1 \\
\text{Fix} (Z) \\
G &:= q \land \Box Z \\
Y &:= 0 \\
\text{Fix} (Y) \\
[Y &:= G \lor \Box Y] \\
Z &:= Y
\end{align*}
\]
Solving \( \lozenge \Box (g_1 = r_1) \) for the Arbiter Example

Applying the above fix-point iterations to the Arbiter example, we obtain:

Note that the obtained strategy, keeps \( g_2 = 0 \) permanently. This suggests that we will have difficulties finding a solution that will maintain

\[ \lozenge \Box (g_1 = r_1) \land \lozenge \Box (g_2 = r_2) \]
Generalized Response (Büchi)

Solving the game for □ ♦ q₁ ∧ ⋯ ∧ □ ♦ qₙ.

\[ \varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} = \mu Y \left( (q_1 \land \Box Z_2) \lor \Box Y \right) + \mu Y \left( (q_2 \land \Box Z_3) \lor \Box Y \right) + \cdots + \mu Y \left( (q_n \land \Box Z_1) \lor \Box Y \right) \]

Iteratively:

For (i ∈ 1..n) do [Z[i] := 1]
Fix (Z[1])

For (i ∈ 1..n) do
\[ Y := 0 \]
Fix (Y)
\[ Y := (q[i] \land \Box Z[i \oplus n 1]) \lor \Box Y \]
\[ Z[i] := Y \]
Return Z[1]
Specification is Unrealizable

Applying the above algorithm to the specification

\[ \square \lozenge (g_1 = r_1) \land \square \lozenge (g_2 = r_2) \]

we find that it fails. Conclusion:

The considered specification is unrealizable

Indeed, without an environment obligation of releasing the resource once it has been granted, the arbiter cannot satisfy any other client.
Property-Based System Design

While the rest of the world seems to be moving in the direction of model-based design (see System-C, UML), some of us persist 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 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.
A Realizable Specification

Assumptions (Constraints on the Environment)

\[ A : \bigwedge_i \left( \overline{r_i} \land (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \land r_i \land g_i \Rightarrow \lozenge \overline{r_i} \right) \]

Guarantees (Expectations from System)

\[ G : \bigwedge_{i \neq j} \neg (g_i \land g_j) \land \bigwedge_i \left( \overline{g_i} \land \left( \begin{array}{l} r_i = g_i \Rightarrow \bigcirc g_i = g_i \\
 r_i \land \overline{g_i} \Rightarrow \lozenge g_i \\
 \overline{r_i} \land g_i \Rightarrow \lozenge \overline{g_i} \end{array} \right) \right) \]

Total Specification

\[ \varphi : A \rightarrow G \]
Assume we are given a set of LTL specifications. We construct a game as follows:

- As $\Theta$ we take all the non-temporal specification parts which relate to the initial state.
- As $\rho_1$ and $\rho_2$, we can take True. A more efficient choice is to include in $\rho_1$ (similarly $\rho_2$) all local limitations on the next values of $X$ (resp. $Y$), such as

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

- We place in $\phi$ all the remaining properties that have not already been included in $\Theta$, $\rho_1$, and $\rho_2$.

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

\[ \bigwedge_i (\overline{r_i} \land (r_i \neq g_i) \Rightarrow (\bigcirc r_i = r_i) \land r_i \land g_i \Rightarrow \Diamond \overline{r_i}) \rightarrow \]

\[ \bigwedge_{i \neq j} \Box \neg(g_i \land g_j) \land \bigwedge_i \left( \overline{g_i} \land \left( \begin{array}{c}
 r_i = g_i \Rightarrow \bigcirc g_i = g_i \land \\
 r_i \land \overline{g_i} \Rightarrow \Diamond g_i \land \\
 \overline{r_i} \land g_i \Rightarrow \Diamond \overline{g_i}
 \end{array} \right) \right) \]

We take the following game components:

\[ X \cup Y : \{ r_i \mid i = 1, \ldots, n \} \cup \{ g_i \mid i = 1, \ldots, n \} \]

\[ \Theta : \bigwedge_i (\overline{r_i} \land \overline{g_i}) \]

\[ \rho_1 : \bigwedge_i ((r_i \neq g_i) \rightarrow (r_i' = r_i)) \]

\[ \rho_2 : \bigwedge_{i \neq j} \neg(g_i' \land g_j') \land \bigwedge_i ((r_i = g_i) \rightarrow (g_i' = g_i)) \]

\[ \varphi : \bigwedge_i (r_i \land g_i \Rightarrow \Diamond \overline{r_i}) \rightarrow \bigwedge_i ((r_i \land \overline{g_i} \Rightarrow \Diamond g_i) \land (\overline{r_i} \land g_i \Rightarrow \Diamond \overline{g_i})) \]
Solving in Polynomial Time a Doubly Exponential Problem

In [1989] Roni Rosner provided a general solution to the Synthesis problem. He showed that any approach that starts with the standard translation from LTL to Büchiautomata, has a doubly exponential lower bound.

One of the messages resulting from the work reported here is

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

For each class of formulas, synthesis can be performed in polynomial time.
Hierarchy of the Temporal Properties

Reactivity
\[ \bigwedge_{i=1}^{k} (\square \Diamond p_i \lor \Diamond \Box q_i) \]

Response
\[ \Box \Diamond p \]

Persistence
\[ \Diamond \Box p \]

Obligation
\[ \bigwedge_{i=1}^{k} (\square p_i \lor \Diamond q_i) \]

Progress

Safety
\[ \square p \]

Guarantee
\[ \Diamond p \]

where \( p, p_i, q, q_i \) are past formulas.
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

$$(□◇ p_1 \land □◇ p_2 \land \cdots \land □◇ p_m) \rightarrow □◇ q_1 \land □◇ q_2 \land \cdots \land □◇ q_n$$

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 so far.

For example, it covers the realizable version of the specification for the Arbiter design.
The Solution

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

\[
\phi = \psi \\
\begin{bmatrix}
Z_1 \\
Z_2 \\
\vdots \\
Z_n
\end{bmatrix}
\begin{bmatrix}
\mu Y \left( \bigvee_{j=1}^{m} \nu X (q_1 \land \square Z_2 \lor \lozenge Y \lor \neg p_j \land \square X) \right) \\
\mu Y \left( \bigvee_{j=1}^{m} \nu X (q_2 \land \square Z_3 \lor \lozenge Y \lor \neg p_j \land \square X) \right) \\
\vdots \\
\mu Y \left( \bigvee_{j=1}^{m} \nu X (q_n \land \square Z_1 \lor \lozenge Y \lor \neg p_j \land \square X) \right)
\end{bmatrix}
\]

where

\[\diamond \phi : \forall X' : \rho_1(V, X') \rightarrow \exists Y' : \rho_2(V, V') \land \phi'(V')\]
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:

There exists 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 $\mathcal{N}^{(2k+1)}$. 
The Semantics of Game Analysis

We can always construct the game tree
Strategies

All strategies can be represented as pruning of the tree at the controllable levels. For example, a strategy for the specification

$$( r \Rightarrow \bigcirc g ) \land ( \overline{r} \Rightarrow \bigcirc \overline{g} )$$
Folding the Tree Into a Finite Graph

In order to be algorithmically tractable, we need to perform the pruning process over the finite graph which generated the game tree. In many cases, this is possible (and leads to memory-less strategy).

![Graph Diagram]
There are cases in which the pruning must depend on the path leading to the current state.

Folding is still possible but may need a longer period.
Controller (Design) Extraction

It remains to show how to extract a winning strategy for the case that a game is winning for player 2.

Let $G : \langle V = X \cup Y, \Theta, \rho_1, \rho_2, \varphi \rangle$ be a given game. A controller for $G$ is an FDS $G_c : \langle V_c, \Theta_c, \rho_c, \emptyset, \emptyset \rangle$, such that:

- $V_c \supseteq V$. That is, $V_c$ extends the set of variables of $G$.
- $\Theta_c \downarrow V = \Theta$. That is, when projecting the initial states of $G_c$ on the variables of $G$, we obtain precisely the initial states of $G$.
- $\rho_c \rightarrow \rho$, where $\rho = \rho_1 \land \rho_2$. That is, if $s_2$ is a $\rho_c$ successor of $s_1$, then $s_2$ is also a $\rho_c$ successor of $s_1$.
- **Player-1 Completeness** — $\rho_c \downarrow_{V, X'} = \rho_1$. That is, a state $s_1 \in \Sigma_c$ has a $\rho_1$-successor $s_2$ iff $s_1$ has a $\rho_c$-successor which agrees with $s_2$ on the valuation of $X$.
- Every infinite run of $G_c$ satisfies the winning condition $\varphi$. 
Example: Extracted Controller for Arbiter

Following is the controller extracted for the Arbiter example:
Interpreting a Controller as a Program

A program (equivalently, a circuit) implementing the extracted controller follows the states that are contained in $G_c$. It has a program counter which ranges over the states of $G_c$.

Assume that control is currently at state $S$ of $G_c$. Let the next values of the input variables be $X = \xi$. Choose a state $S'$ which is a $\rho_c$-successor of $S$, and such that $S'[X] = \xi$. By the requirement of Player-1 Completeness, there always exists such a successor.

The actions of the program is to output the values $\eta$ such that $S'[Y] = \eta$, and to move to state $S'$. 
Computing a Controller for the Winning Condition $\square p$

The winning states in a game with a winning condition $\square p$ are given by:

$$\text{win} = \nu Z. p \land \lozenge Z$$

The full controller extraction algorithm can be given by the following program:

$$\begin{align*}
Z & := 1 \\
\text{Fix} (Z) & \left[ Z := p \land \lozenge Z \right] \\
\text{if } (\Theta \land \neg Z) \neq 0 & \text{ then Print "Specification is unrealizable"} \\
\text{else} & \\
\Theta_c & := \Theta \\
\rho_c & := Z \land \rho \land Z' \\
\end{align*}$$

**Claim 10.** If $s$ is a winning state of a $\square p$-game, then $s \models p$, and player 2 can force the game to move from $s$ to a successor which is also a winning state.
Computing A Controller for the Winning Condition $\diamondsuit q$

The winning states in a game with a winning condition $\diamondsuit p$ are given by:

$$\text{win} = \mu Y. q \lor \Box Y$$

The full controller extraction algorithm can be given by the following program:

$$Y := q; \quad r := 0; \quad U[0] := q$$

Fix $(Y)$

$$\begin{array}{l}
Y := q \lor \Box Y; \quad r := r + 1; \quad U[r] := Y \\
\text{if } (\Theta \land \neg Y) \neq 0 \text{ then} \\
\quad \text{Print } "\text{Specification is unrealizable}" \\
\text{else} \\
\quad \Theta_c := \Theta \\
\quad \rho_c := 0; \quad prev := U[0] \\
\quad \text{for } i \in 1 \ldots r \text{ do} \\
\quad \quad \begin{array}{l}
\rho_c := \rho_c \lor (U[i] \land \neg prev) \land \rho \land prev'; \quad prev := prev \lor U[i]
\end{array}
\end{array}$$

Claim 11. Every winning state $s$ in a $(\diamondsuit q)$-game is associated with a natural rank $r(s) \geq 0$, such that if $r(s) = 0$ then $s \models q$, and if $r(s) > 0$, then player 2 can force the game to move from $s$ to a winning successor with a lower rank.
Computing A Controller for the Winning Condition \( \square \lozenge q \)

The winning states in a game with a winning condition \( \square \lozenge p \) are given by:

\[
\text{win} = \nu Z \mu Y. (q \land \lozenge Z) \lor \lozenge Y
\]

The full controller extraction algorithm can be given by the following program:

\[
\begin{align*}
Z & := 1 \\
\text{Fix} (Z) & \quad \begin{cases}
Y := q \land \lozenge Z; \\
r := 0; \\
U[0] := Y
\end{cases}
\\n\text{Fix} (Y) & \quad \begin{cases}
Y := q \lor \lozenge Y; \\
r := r + 1; \\
U[r] := Y
\end{cases}
\end{align*}
\]

if \( (\Theta \land \neg Z) \neq 0 \) then

Print "Specification is unrealizable"

else

\[
\begin{align*}
\Theta_c & := \Theta \\
\rho_c & := U[0] \land \rho \land Z'; \\
\text{prev} & := U[0] \\
\text{for } i \in 1 \ldots r \text{ do} & \quad \begin{cases}
\rho_c := \rho_c \lor (U[i] \land \neg \text{prev}) \land \rho \land \text{prev'}; \\
\text{prev} := \text{prev} \lor U[i]
\end{cases}
\end{align*}
\]

Claim 12. Every winning state \( s \) in a \( \square \lozenge q \)-game is associated with a natural rank \( r(s) \), such that player 2 can force the game to move from \( s \) to a winning successor \( s' \) where either \( r(s) = 0 \) and \( s \models q \), or \( r(s) > r(s') \).