Synthesis of Reactive(1) Designs*

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

[1] EPFL - I&C - MTC, 1015, Lausanne, Switzerland
firstname.lastname@epfl.ch

[2] Department of Computer Science, Weizmann Institute of Science, Rehovot, 76100, Israel
firstname.lastname@weizmann.ac.il

[3] Department of Computer Science, Ben Gurion University, Beer-Sheva, Israel
saary@cs.bgu.ac.il

Abstract. We consider the problem of synthesizing digital designs from their LTL specification. In spite of the theoretical double exponential lower bound for the general case, we show that for many expressive specifications of hardware designs the problem can be solved in time N^3, where N is the size of the state space of the design. We describe the context of the problem, as part of the Prosyd European Project which aims to provide a property-based development flow for hardware designs. Within this project, synthesis plays an important role, first in order to check whether a given specification is realizable, and then for synthesizing part of the developed system. The class of LTL formulas considered is that of Generalized Reactivity(1) (generalized Streett(1)) formulas, i.e., formulas of the form:

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

where each p_i, q_i is a boolean combination of atomic propositions. We also consider the more general case in which each p_i, q_i is an arbitrary past LTL formula over atomic propositions.

For this class of formulas, we present an N^3-time algorithm which checks whether such a formula is realizable, i.e., there exists a circuit which satisfies the formula under any set of inputs provided by the environment. In the case that the specification is realizable, the algorithm proceeds to construct an automaton which represents one of the possible implementing circuits. The automaton is computed and presented symbolically.

1 Introduction

One of the most ambitious and challenging problems in reactive systems construction is the automatic synthesis of programs and (digital) designs from logical specifications. First identified as Church's problem [Chu63], several methods have been proposed for its solution ([BL69], [Rab72]). The two prevalent approaches to solving the synthesis problem were by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person game. In these preliminary studies of the problem,

* This research was supported in part by the Israel Science Foundation (grant no.106/02-1), European community project Prosyd, the John von-Neumann Minerva center for Verification of Reactive Systems, NSF grant CCR-0205571, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 364–380, 2006.

the logical specification that the synthesized system should satisfy was given as an S1S formula.

This problem has been considered again in [PR89a] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts ([CE81], [MW84]) to synthesize programs from temporal specification which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. The method proposed in [PR89a] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then determinized into a deterministic Rabin automaton. This double translation may reach complexity of double exponent in the size of φ. Once the Rabin automaton is obtained, the game can be solved in time $n^{O(k)}$, where n is the number of states of the automaton and k is the number of accepting pairs.

The high complexity established in [PR89a] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Representative cases are the work in [AMPS98] which presents (besides the generalization to real time) efficient polynomial solutions (N^2) to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulas $\Box\, p$, $\Diamond\, q$, $\Box\Diamond\, p$, or $\Diamond\Box\, q$. A more recent paper is [AT04] which presents efficient synthesis approaches for the LTL fragment consisting of a boolean combinations of formulas of the form $\Box\, p$.

This paper can be viewed as a generalization of the results of [AMPS98] and [AT04] into the wider class of *generalized Reactivity*(1) formulas (GR(1)), i.e. formulas of the form

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \quad \rightarrow \quad (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n) \tag{1}$$

Following the developments in [KPP05], we show how any synthesis problem whose specification is a GR(1) formula can be solved in time N^3, where N is the size of the state space of the design. Furthermore, we present a (symbolic) algorithm for extracting a design (program) which implements the specification. We make an argument that the class of GR(1) formulas is sufficiently expressive to provide complete specifications of many designs.

This work has been developed as part of the Prosyd project (see www.prosyd.org) which aims at the development of a methodology and a tool suit for the property-based construction of digital circuits from their temporal specification. Within the prosyd project, synthesis techniques are applied to check first whether a set of properties is *realizable*, and then to automatically produce digital designs of smaller units.

2 Preliminaries

2.1 Linear Temporal Logic

We assume a countable set of Boolean variables (propositions) $\mathcal{V}$. LTL formulas are constructed as follows.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

As usual we denote $\neg(\neg\varphi \vee \neg\psi)$ by $\varphi \wedge \psi$, $\mathrm{T}\,\mathcal{U}\varphi$ by $\diamondsuit\, \varphi$ and $\neg\diamondsuit\neg\varphi$ by $\square\,\varphi$. A formula that does not include temporal operators is a *Boolean formula*.

A model σ for a formula φ is an infinite sequence of truth assignments to propositions. Namely, if P' is the set of propositions appearing in φ, then for every finite set P such that $P' \subseteq P$, a word in $(2^P)^\omega$ is a model. We denote by $\sigma(i)$ the set of propositions at position i, that is $\sigma = \sigma(0), \sigma(1), \ldots$. We present an inductive definition of when a formula holds in model σ at position i.

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma(i)$.
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$
- $\sigma, i \models \varphi \vee \psi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \psi$
- $\sigma, i \models \bigcirc\,\varphi$ iff $\sigma, i+1 \models \varphi$
- $\sigma, i \models \varphi\mathcal{U}\psi$ iff there exists $k \geq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $i \leq j < k$

For a formula φ and a position $j \geq 0$ such that $\sigma, j \models \varphi$, we say that φ *holds at position* j of σ. If $\sigma, 0 \models \varphi$ we say that φ *holds* on σ and denote it by $\sigma \models \varphi$. A set of models L satisfies φ, denoted $L \models \varphi$, if every model in L satisfies φ.

We are interested in the question of *realizability* of LTL specifications [PR89b]. Assume two sets of variables $\mathcal{X}$ and $\mathcal{Y}$. Intuitively $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of system variables. With no loss of generality, we assume that all variables are Boolean. Obviously, the more general case that $\mathcal{X}$ and $\mathcal{Y}$ range over arbitrary finite domains can be reduced to the Boolean case. *Realizability* amounts to checking whether there exists an *open controller* that satisfies the specification. Such a controller can be represented as an automaton which, at any step, inputs values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables. Below we formalize the notion of checking realizability and *synthesis*, namely, the construction of such controllers.

Realizability for LTL specifications is 2EXPTIME-complete [PR90]. We are interested in a subset of LTL for which we solve realizability and synthesis in polynomial time. The specifications we consider are of the form $\varphi = \varphi_e \rightarrow \varphi_s$. We require that φ_α for $\alpha \in \{e, s\}$ can be rewritten as a conjunction of the following parts.

- φ_i^α - a Boolean formula which characterizes the initial states of the implementation.
- φ_t^α - a formula of the form $\bigwedge_{i \in I} \square\, B_i$ where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc\, v$ where $v \in \mathcal{X}$ if $\alpha = e$, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
- φ_g^α - a formula of the form $\bigwedge_{i \in I} \square\diamondsuit\, B_i$ where each B_i is a Boolean formula.

It turns out that most of the specifications written in practice can be rewritten to this format[1]. In Section 7 we discuss also cases where the formulas φ_g^α have also sub-formulas of the form $\square(p \rightarrow \diamondsuit\, q)$ where p and q are Boolean formulas, and additional cases which can be converted to the GR(1) format.

[1] In practice, the specification is usually given in this format. The specification is a collection of assumptions and requirements with the semantics that all assumptions imply all requirements. Every assumption or requirement is usually of a very simple formula similar to the required form.

2.2 Game Structures

We reduce the realizability problem of an LTL formula to the decision of winner in games. We consider two-player games played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. Formally, we have the following.

A *game structure* (GS) $G : \langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi\rangle$ consists of the following components.

- $V = \{u_1, \ldots, u_n\}$: A finite set of typed *state variables* over finite domains. With no loss of generality, we assume they are all Boolean. We define a *state* s to be an interpretation of V, assigning to each variable $u \in V$ a value $s[u] \in \{0, 1\}$. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to Boolean expressions over V in the usual way. An *assertion* is a Boolean formula over V. A state s satisfies an assertion φ denoted $s \models \varphi$, if $s[\varphi] = \mathbf{true}$. We say that s is a φ-state if $s \models \varphi$.
- $\mathcal{X} \subseteq V$ is a set of *input variables*. These are variables controlled by the environment. Let D_X denote the possible valuations to variables in $\mathcal{X}$.
- $\mathcal{Y} = V \backslash \mathcal{X}$ is a set of *output variables*. These are variables controlled by the system. Let D_Y denote the possible valuations for the variables in $\mathcal{Y}$.
- Θ is the initial condition. This is an assertion characterizing all the initial states of G. A state is called *initial* if it satisfies Θ.
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the transition relation of the environment. This is an assertion, relating a state $s \in \Sigma$ to a possible next input value $\boldsymbol{\xi}' \in D_X$, by referring to unprimed copies of $\mathcal{X}$ and $\mathcal{Y}$ and primed copies of $\mathcal{X}$. The transition relation ρ_e identifies valuation $\boldsymbol{\xi}' \in D_X$ as a possible *input* in state s if $(s, \boldsymbol{\xi}') \models \rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ where $(s, \boldsymbol{\xi}')$ is the joint interpretation which interprets $u \in V$ as $s[u]$ and for $v \in \mathcal{X}$ interprets v' as $\boldsymbol{\xi}'[v]$.
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the system. This is an assertion, relating a state $s \in \Sigma$ and an input value $\boldsymbol{\xi}' \in D_X$ to a next output value $\boldsymbol{\eta}' \in D_Y$, by referring to primed and unprimed copies of V. The transition relation ρ_s identifies a valuation $\boldsymbol{\eta}' \in D_Y$ as a possible *output* in state s reading input $\boldsymbol{\xi}'$ if $(s, \boldsymbol{\xi}', \boldsymbol{\eta}') \models \rho_s(V, V')$ where $(s, \boldsymbol{\xi}', \boldsymbol{\eta}')$ is the joint interpretation which interprets $u \in \mathcal{X}$ as $s[u]$, u' as $\boldsymbol{\xi}'[u]$, and similarly for $v \in \mathcal{Y}$.
- φ is the winning condition, given by an LTL formula.

For two states s and s' of G, s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We freely switch between $(s, \boldsymbol{\xi}') \models \rho_e$ and $\rho_e(s, \boldsymbol{\xi}') = 1$ and similarly for ρ_s. A *play* σ of G is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* namely $s_0 \models \Theta$, and *consecution* namely, for each $j \geq 0$, s_{j+1} is a successor of s_j. Let G be an GS and σ be a play of G. From a state s, the environment chooses an input $\boldsymbol{\xi}' \in D_X$ such that $\rho_e(s, \boldsymbol{\xi}') = 1$ and the system chooses an output $\boldsymbol{\eta}' \in D_Y$ such that $\rho_s(s, \boldsymbol{\xi}', \boldsymbol{\eta}') = \rho_s(s, s') = 1$.

A play σ is *winning for the system* if it is infinite and it satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is a partial function $f : \Sigma^+ \times D_X \mapsto D_Y$ such that if $\sigma = s_0, \ldots s_n$ then for every $\boldsymbol{\xi}' \in D_X$ such that $\rho_e(s_n, \boldsymbol{\xi}') = 1$ we have $\rho_s(s_n, \boldsymbol{\xi}', f(\sigma, \boldsymbol{\xi}')) = 1$. Let f be a strategy for the system, and $s_0 \in \Sigma$. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \ldots, s_i, s_{i+1}[\mathcal{X}]) = s_{i+1}[\mathcal{Y}]$, where

$s_{i+1}[\mathcal{X}]$ and $s_{i+1}[\mathcal{Y}]$ are the restrictions of s_{i+1} to variable sets $\mathcal{X}$ and $\mathcal{Y}$, respectively. Strategy f is *winning* for the system from state $s \in \Sigma_G$ if all s-plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_s the set of states from which there exists a winning strategy for the system. A *strategy* for player environment, *winning strategy*, and the *winning set* W_e are defined dually. A GS G is said to be *winning* for the system if all initial states are winning for the system.

Given an LTL specification $\varphi_e \rightarrow \varphi_s$ as explained above and sets of input and output variables $\mathcal{X}$ and $\mathcal{Y}$ we construct a GS as follows. Let $\varphi_\alpha = \varphi_i^\alpha \wedge \varphi_t^\alpha \wedge \varphi_g^\alpha$ for $\alpha \in \{e, s\}$. Then, for Θ we take $\varphi_i^e \wedge \varphi_i^s$. Let $\varphi_t^\alpha = \bigwedge_{i \in I} \square B_i$, then $\rho_\alpha = \bigwedge_{i \in I} \tau(B_i)$, where the translation τ replaces each instance of $\bigcirc v$ by v'. Finally, we set $\varphi = \varphi_g^e \rightarrow \varphi_g^s$. We *solve* the game, attempting to decide whether the game is winning for the environment or the system. If the environment is winning the specification is *unrealizable*. If the system is winning, we *synthesize* a winning strategy which is a *working implementation* for the system as explained in Section 4.

2.3 Fair Discrete Systems

We present implementations as a special case of*fair discrete systems* (FDS) [KP00]. An FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of Boolean variables. We define a *state* s to be an interpretation of V. Denote by Σ the set of all states. Assertions over V and satisfaction of assertions are defined like in games.
- Θ : The *initial condition*. This is an assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$.
- $\mathcal{J} = \{J_1, \ldots, J_m\}$: A set of *justice requirements* (weak fairness). Each requirement $J \in \mathcal{J}$ is an assertion which is intended to hold infinitely many times in every computation.
- $\mathcal{C} = \{(p_1, q_1), \ldots, (p_n, q_n)\}$: A set of *compassion requirements* (strong fairness). Each requirement $(p, q) \in \mathcal{C}$ consists of a pair of assertions, such that if a computation contains infinitely many p-states, it should also hold infinitely many q-states.

We define a *run* of the FDS $\mathcal{D}$ to be a maximal sequence of states $\sigma : s_0, s_1, ...,$ satisfying the requirements of

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

The sequence σ being maximal means that either σ is infinite, or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor.

A run σ is defined to be a *computation* of $\mathcal{D}$ if it is infinite and satisfies the following additional requirements:

- *Justice:* For each $J \in \mathcal{J}$, σ contains infinitely many J-positions, i.e. positions $j \geq 0$, such that $s_j \models J$.
- *Compassion:* For each $(p, q) \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

We say that an FDS $\mathcal{D}$ *implements* specification φ if every run of $\mathcal{D}$ is infinite, and every computation of $\mathcal{D}$ satisfies φ. An FDS is said to be *fairness-free* if $\mathcal{J} = \mathcal{C} = \emptyset$. It is called a *just transition system* (JDS) if $\mathcal{C} = \emptyset$.

In general, we use FDS's in order to formalize reactive systems. When we formalize concurrent systems which communicate by shared variables as well as most digital designs, the ensuing formal model is that of a JDS (i.e., compassion-free). Compassion is needed only in the case that the program uses built-in synchronization constructs such as semaphores or synchronous communication.

For every FDS, there exists an LTL formula $\varphi_{\mathcal{D}}$, called the *temporal semantics* of $\mathcal{D}$ which fully characterizes the computations of $\mathcal{D}$. It can be written as:

$$\varphi_{\mathcal{D}}: \quad \Theta \wedge \Box(\rho(V, \bigcirc V)) \wedge \bigwedge_{J\in\mathcal{J}} \Box\Diamond J \wedge \bigwedge_{(p,q)\in\mathcal{C}} (\Box\Diamond p \rightarrow \Box\Diamond q)$$

where $\rho(V, \bigcirc V)$ is the formula obtained from $\rho(V, V')$ by replacing each instance of primed variable x' by the LTL formula $\bigcirc x$.
Note that in the case that $\mathcal{D}$ is compassion-free (i.e., it is a JDS), then its temporal semantics has the form

$$\varphi_{\mathcal{D}}: \quad \Theta \wedge \Box(\rho(V, \bigcirc V)) \wedge \bigwedge_{J\in\mathcal{J}} \Box\Diamond J$$

It follows that the class of specifications we consider in this paper, as explained at the end of Subsection 2.1, have the form $\varphi = \varphi_e \rightarrow \varphi_s$ where each φ_α, for $\alpha \in \{e, s\}$, is the temporal semantics of an JDS. Thus, if the specification can be realized by an environment which is a JDS and a system which is a JDS (in particular, if none of them requires compassion for their implementation), then the class of specifications we consider here are as general as necessary. Note in particular, that hardware designs rarely assume compassion (strong fairness) as a built-in construct. Thus, we expect most specifications to be realized by hardware designs to fall in the class of GR(1).

3 μ-Calculus and Games

In [KPP05], we consider the case of GR(1) games (called there *generalized Streett(1)* games). In these games the winning condition is an implication between conjunctions of recurrence formulas ($\Box\Diamond\varphi$ where φ is a Boolean formula). These are exactly the types of goals in the games we defined in Section 2. We show how to solve such games in cubic time [KPP05]. We re-explain here how to compute the winning regions of each of the players and explain how to use the algorithm to extract a winning strategy. We start with a definition of μ-calculus over game structures. We give the μ-calculus formula that characterizes the set of winning states of the system. We explain how we construct from this μ-calculus formula an algorithm to compute the set of winning states. Finally, by saving intermediate values in the computation, we can construct a winning strategy and synthesize an FDS that implements the goal.

3.1 μ-Calculus over Games Structures

We define μ-calculus [Koz83] over game structures. Let G: $\langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi\rangle$ be a GS. For every variable $v \in V$ the formulas v and $\neg v$ are *atomic formulas*. Let $Var =$

$\{X, Y, \ldots\}$ be a set of *relational variables*. The μ-calculus formulas are constructed as follows.

$$\varphi ::= v \mid \neg v \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\!\!\!\!\!\!\Diamond\, \varphi \mid \bigcirc\!\!\!\!\!\!\Box\, \varphi \mid \mu X \varphi \mid \nu X \varphi$$

A formula ψ is interpreted as the set of G-states in Σ in which ψ is true. We write such set of states as $[[\psi]]^e_G$ where G is the GS and $e : Var \rightarrow 2^\Sigma$ is an *environment*. The environment assigns to each relational variable a subset of Σ. We denote by $e[X \leftarrow S]$ the environment such that $e[X \leftarrow S](X) = S$ and $e[X \leftarrow S](Y) = e(Y)$ for $Y \neq X$. The set $[[\psi]]^e_G$ is defined inductively as follows[2].

- $[[v]]^e_G = \{s \in \Sigma \mid s[v] = 1\}$
- $[[\neg v]]^e_G = \{s \in \Sigma \mid s[v] = 0\}$
- $[[X]]^e_G = e(X)$
- $[[\varphi \vee \psi]]^e_G = [[\varphi]]^e_G \cup [[\psi]]^e_G$
- $[[\varphi \wedge \psi]]^e_G = [[\varphi]]^e_G \cap [[\psi]]^e_G$
- $[[\bigcirc\!\!\!\!\!\!\Diamond\, \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall \boldsymbol{x}',\ (s, \boldsymbol{x}') \models \rho_e \rightarrow \exists \boldsymbol{y}' \text{ such that } (s, \boldsymbol{x}', \boldsymbol{y}') \models \rho_s \\ \text{and } (\boldsymbol{x}', \boldsymbol{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\bigcirc\!\!\!\!\!\!\Diamond\, \varphi]]^e_G$ if the system can force the play to reach a state in $[[\varphi]]^e_G$. That is, regardless of how the environment moves from s, the system can choose an appropriate move into $[[\varphi]]^e_G$.
- $[[\bigcirc\!\!\!\!\!\!\Box\, \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \exists \boldsymbol{x}' \text{ such that } (s, \boldsymbol{x}') \models \rho_e \text{ and} \\ \forall \boldsymbol{y}',\ (s, \boldsymbol{x}', \boldsymbol{y}') \models \rho_s \rightarrow (\boldsymbol{x}', \boldsymbol{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\bigcirc\!\!\!\!\!\!\Box\, \varphi]]^e_G$ if the environment can force the play to reach a state in $[[\varphi]]^e_G$. As the environment moves first, it chooses an input $\boldsymbol{x}' \in X$ such that for all choices of the system the successor s is in $[[\varphi]]^e_G$.
- $[[\mu X \varphi]]^e_G = \cup_i S_i$ where $S_0 = \emptyset$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$
- $[[\nu X \varphi]]^e_G = \cap_i S_i$ where $S_0 = \Sigma$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$

When all the variables in φ are bound by either μ or ν the initial environment is not important and we simply write $[[\varphi]]_G$. In case that G is clear from the context we write $[[\varphi]]$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. A μ-calculus formula defines a symbolic algorithm for computing $[[\varphi]]$ [EL86]. For a μ-calculus formula of alternation depth k, the run time of this algorithm is $O(|\Sigma|^k)$. For a full exposition of μ-calculus we refer the reader to [Eme97]. We often abuse notations and write a μ-calculus formula φ instead of the set $[[\varphi]]$.

In some cases, instead of using a very complex formula, it may be more readable to use *vector notation* as in Equation (2) below.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \begin{bmatrix} \mu Y (\bigcirc\!\!\!\!\!\!\Diamond\, Y \ \vee \ p \wedge \bigcirc\!\!\!\!\!\!\Diamond\, Z_2) \\ \mu Y (\bigcirc\!\!\!\!\!\!\Diamond\, Y \ \vee \ q \wedge \bigcirc\!\!\!\!\!\!\Diamond\, Z_1) \end{bmatrix} \qquad (2)$$

Such a formula, may be viewed as the mutual fixpoint of the variables Z_1 and Z_2 or equivalently as an equal formula where a single variable Z replaces both Z_1 and Z_2 and ranges over pairs of states [Lic91]. The formula above characterizes the set of states from

[2] Only for finite game structures.

which system can force the game to visit p-states infinitely often and q-states infinitely often. We can characterize the same set of states by the following 'normal' formula[3].

$$\varphi = \nu Z \left([\mu Y(\circleddiamond Y \ \vee \ p \wedge \circleddiamond Z)] \ \wedge \ [\mu Y(\circleddiamond Y \ \vee \ q \wedge \circleddiamond Z)] \right).$$

3.2 Solving GR(1) Games

Let G be a game where the winning condition is of the following form.

$$\varphi = \bigwedge_{i=1}^{m} \Box \Diamond J_i^1 \rightarrow \bigwedge_{j=1}^{n} \Box \Diamond J_j^2$$

Here J_i^1 and J_j^2 are sets of Boolean formulas. In [KPP05] we term these games as generalized Streett(1) games and provide the following μ-calculus formula to solve them. Let $j \oplus 1 = (j \ mod \ n) + 1$.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_1^2 \wedge \circleddiamond Z_2 \ \vee \ \circleddiamond Y \ \vee \ \neg J_i^1 \wedge \circleddiamond X) \right) \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_2^2 \wedge \circleddiamond Z_3 \ \vee \ \circleddiamond Y \ \vee \ \neg J_i^1 \wedge \circleddiamond X) \right) \\ \vdots \\ \vdots \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_n^2 \wedge \circleddiamond Z_1 \ \vee \ \circleddiamond Y \ \vee \ \neg J_i^1 \wedge \circleddiamond X) \right) \end{bmatrix} \tag{3}$$

Intuitively, for $j \in [1..n]$ and $i \in [1..m]$ the greatest fixpoint $\nu X (J_j^2 \wedge \circleddiamond Z_{j \oplus 1} \ \vee \ \circleddiamond Y \ \vee \ \neg J_i^1 \wedge \circleddiamond X)$ characterizes the set of states from which the system can force the play either to stay indefinitely in $\neg J_i^1$ states (thus violating the left hand side of the implication) or in a finite number of steps reach a state in the set $J_j^2 \wedge \circleddiamond Z_{j \oplus 1} \vee \circleddiamond Y$. The two outer fixpoints make sure that the system wins from the set $J_j^2 \wedge \circleddiamond Z_{j \oplus 1} \ \vee \ \circleddiamond Y$. The least fixpoint μY makes sure that the unconstrained phase of a play represented by the disjunct $\circleddiamond Y$ is finite and ends in a $J_j^2 \wedge \circleddiamond Z_{j \oplus 1}$ state. Finally, the greatest fixpoint νZ_j is responsible for ensuring that, after visiting J_j^2, we can loop and visit $J_{j \oplus 1}^2$ and so on. By the cyclic dependence of the outermost greatest fixpoint, either all the sets in J_j^2 are visited or getting stuck in some inner greatest fixpoint, where some J_i^1 is visited only finitely many times.

We include in Fig. 1 a (slightly simplified) code of the implementation of this μ-calculus formula in TLV (see Section 5). We denote J_i^α for $\alpha \in \{1, 2\}$ by $Ji(i, \alpha)$ and $\circleddiamond$ by *cox*. We denote conjunction, disjunction, and negation by &, |, and ! respectively. A GreatestFixpoint loop on variable u starts by setting the initial value of u to the set of all states and a LeastFixpoint loop over u starts by setting u to the empty set of states. For both types of fixpoints, the loop terminates if two successive values of u are the same. The greatest fixpoint `GreatestFixpoint(x <= z)`, means that the initial

[3] This does not suggest a canonical translation from vector formulas to plain formulas. The same translation works for the formula in Equation (3) below. Note that the formula in Equation (2) and the formula in Equation (3) have a very similar structure.

```
Func winm(m, n);
  GreatestFixpoint(z)
    For (j in 1...n)
      Let r := 1;
      LeastFixpoint (y)
        Let start := Ji(j,2) & cox(z) | cox(y);
        Let y := 0;
        For (i in 1...m)
          GreatestFixpoint (x <= z)
            Let x := start | !Ji(i,1) & cox(x);
          End -- GreatestFixpoint (x)
          Let x[j][r][i] := x; // store values of x
          Let y := y | x;
        End -- For (i in 1...m)
        Let y[j][r] := y; // store values of y
        Let r := r + 1;
      End -- LeastFixpoint (y)
      Let z := y;
      Let maxr[j] := r - 1;
    End -- For (j in 1...m)
  End -- GreatestFixpoint (z)
  Return z;
End -- Func winm(m, n);
```

Fig. 1. TLV implementation of Equation (3)

value of x is z instead of the universal set of all states. We use the sets $y[j][r]$ and their subsets $x[j][r][i]$ to define n strategies for the system. The strategy f_j is defined on the states in Z_j. We show that the strategy f_j either forces the play to visit J_j^2 and then proceed to $Z_{j\oplus 1}$, or eventually avoid some J_i^1. We show that by combining these strategies, either the system switches strategies infinitely many times and ensures that the play be winning according to right hand side of the implication or eventually uses a fixed strategy ensuring that the play does not satisfy the left hand side of the implication. Essentially, the strategies are "go to $y[j][r]$ for minimal r" until getting to a J_j^2 state and then switch to strategy $j \oplus 1$ or "stay in $x[j][r][i]$".

It follows that we can solve realizability of LTL formulas in the form that interests us in polynomial (cubic) time.

Theorem 1. [KPP05] *Given sets of variables $\mathcal{X}$, $\mathcal{Y}$ whose set of possible valuations is Σ and an* LTL *formula φ with m and n conjuncts, we can determine using a symbolic algorithm whether φ is realizable in time proportional to $(nm|\Sigma|)^3$.*

4 Synthesis

We show how to use the intermediate values in the computation of the fixpoint to produce an FDS that implements φ. The FDS basically follows the strategies explained above.

Let $\mathcal{X}$, $\mathcal{Y}$, and φ be as above. Let G: $\langle V, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \Theta, \varphi_g\rangle$ be the GS defined by $\mathcal{X}$, $\mathcal{Y}$, and φ (where $V = \mathcal{X} \cup \mathcal{Y}$). We construct the following fairness-free FDS. Let

$\mathcal{D}: \langle V_{\mathcal{D}}, \mathcal{X}, \mathcal{Y}_{\mathcal{D}}, \varTheta_{\mathcal{D}}, \rho\rangle$ where $V_{\mathcal{D}} = V \cup \{jx\}$ and jx ranges over $[1..n]$, $\mathcal{Y}_{\mathcal{D}} = \mathcal{Y} \cup \{jx\}$, $\varTheta_{\mathcal{D}} = \varTheta \wedge (jx = 1)$. The variable jx is used to store internally which strategy should be applied. The transition ρ is $\rho_1 \vee \rho_2 \vee \rho_3$ where ρ_1, ρ_2, and ρ_3 are defined as follows.

Transition ρ_1 is the transition taken when a J^2_j state is reached and we change strategy from f_j to $f_{j\oplus 1}$. Accordingly, all the disjuncts in ρ_1 change jx. Transition ρ_2 is the transition taken in the case that we can get closer to a J^2_j state. These transitions go from states in some set $y[j][r]$ to states in the set $y[j][r']$ where $r' < r$. We take care to apply this transition only to states s for which $r > 1$ is the minimal index such that $s \in y[j][r]$. Transition ρ_3 is the transition taken from states $s \in x[j][r][i]$ such that $s \models \neg J^1_i$ and the transition takes us back to states in $x[j][r][i]$. Repeating such a transition forever will also lead to a legitimate computation because it violates the environment requirement of infinitely many visits to J^1_i-states. Again, we take care to apply this transition only to states for which (r, i) are the (lexicographically) minimal indices such that $s \in x[j][r][i]$.

Let $y[j][< r]$ denote the set $\bigcup_{l\in[1..r-1]} y[j][l]$. We write $(r', i') \prec (r, i)$ to denote that the pair (r', i') is lexicographically smaller than the pair (r, i). That is, either $r' < r$ or $r' = r$ and $i' < i$. Let $x[j][\prec(r, i)]$ denote the set $\bigcup_{(r',i')\prec(r,i)} x[j][r'][i']$. The transitions are defined as follows.

$$\begin{aligned}
\rho_1 \quad &= \bigvee_{j\in[1..n]} (jx{=}j) \wedge z \wedge J^2_j \wedge \rho_e \wedge \rho_s \wedge z' \wedge (jx'{=}j{\oplus}1)\\
\rho_2(j) &= \bigvee_{r>1} y[j][r] \wedge \neg y[j][< r] \wedge \rho_e \wedge \rho_s \wedge y'[j][< r]\\
\rho_2 \quad &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_2(j)\\
\rho_3(j) &= \bigvee_{r} \bigvee_{i\in[1..m]} x[j][r][i] \wedge \neg x[j][\prec(r, i)] \wedge \neg J^1_i \wedge \rho_e \wedge \rho_s \wedge x'[j][r][i]\\
\rho_3 \quad &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_3(j)
\end{aligned}$$

The conjuncts $\neg y[j][< r]$ and $\neg x[j][\prec(r, i)]$ appearing in transitions $\rho_2(j)$ and $\rho_3(j)$ ensure the minimality of the indices to which these transitions are respectively applied.

Notice that the above transitions can be computed symbolically. We include below the TLV code that symbolically constructs the transition relation of the synthesized FDS and places it in `trans`. We denote the conjunction of ρ_e and ρ_s by `trans12`.

```
To symb_strategy;
  Let trans := 0;
  For (j in 1...n)
    Let jp1 := (j mod n) + 1;
    Let trans := trans | (jx=j) & z & Ji(j,2) & trans12 &
                          next(z) & (next(jx)=jp1);
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := y[j][1];
    For (r in 2...maxr[j])
      Let trans := trans | (jx=j) & y[j][r] & !low &
                      trans12 & next(low) & (next(jx)=j);
      Let low := low | y[j][r];
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := 0;
    For (r in 2...maxr[j])
      For (i in 1...m)
        Let trans := trans | (jx=j) & x[j][r][i] & !low
                                   & !ji(i,1) & trans12 &
                         next(x[j][r][i]) & (next(jx)=j);
        Let low := low | x[j][r][i];
      End -- For (i in 1...m)
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
End -- To symb_strategy;
```

4.1 Minimizing the Strategy

We have created an FDS that implements an LTL goal φ. The set of variables of this FDS includes the given set of input and output variables as well as a 'memory' variable jx. We have quite a liberal policy of choosing the next successor in the case of a visit to J_j^2. We simply choose some successor in the winning set. Here we minimize (symbolically) the resulting FDS. A necessary condition for the soundness of this minimization is that the specification be insensitive to stuttering[4]

Notice, that our FDS is deterministic. For every state and every possible assignment to the variables in $\mathcal{X} \cup \mathcal{Y}$ there exists at most one successor state with this assignment. Thus, removing transitions seems to be of lesser importance. We concentrate on removing redundant states.

As we are using the given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ the only possible candidate states for merging are states that agree on the values of variables in $\mathcal{X} \cup \mathcal{Y}$ and disagree on the value of jx. If we find two states s and s' such that $\rho(s, s')$, $s[\mathcal{X} \cup \mathcal{Y}] = s'[\mathcal{X} \cup \mathcal{Y}]$, and $s'[jx] = s[jx] \oplus 1$, we remove state s. We direct all its incoming arrows to s' and remove its outgoing arrows. Intuitively, we can do that because for every computation that passes through s there exists a computation that stutters once in s (due to the assumption of stuttering insensitivity). This modified computation passes from s to s' and still satisfies all the requirements (we know that stuttering in s is allowed because there exists a transition to s' which agrees with s on all variables).

As mentioned this minimization is performed symbolically. As we discuss in Section 5, it turns out that the minimization actually increases the size of the resulting BDDs.

[4] A specification is insensitive to stuttering if the result of doubling a letter (or replacing a double occurrence by a single occurrence) in a model is still a model. The specifications we consider are allowed to use the next operator, thus they can be sensitive to stuttering. A specification that requires that in some case an immediate response be made would be sensitive to stuttering.

It seems to us that for practical reasons we may want to keep the size of BDDs minimal rather than minimize the automaton. The symbolic implementation of the minimization is given below. The transition *obseq* includes all possible assignments to V and V' such that all variables except jx maintain their values. It is enough to consider the transitions from j to $j \oplus 1$ for all j and then from n to j for all j to remove all redundant states. This is because the original transition just allows to increase jx by one.

```
For (j in 1..n)
  Let nextj := (j mod n)+1;
  reduce(j,nextj);
End -- For (j in 1..n)

For (j in 1..n-1)
  reduce(n,j)
End -- For (j in 1..n-1)

Func reduce(j,k)
  Let idle := trans & obseq & jx=j & next(jx)=k;
  Let states := idle forsome next(V);
  Let add_trans :=
      ((trans & next(states) & next(jx)=j) forsome jx) &
      next(jx)=k;
  Let rem_trans := next(states) & next(jx)=j1 |
                   states & jx=j1;
  Let add_init := ((init & states & jx=j1) forsome jx) &
                  jx=k;
  Let rem_init := states & jx=j;
  Let trans := (trans & !rem_trans) | add_trans;
  Let init := (init & !rem_init) | add_init;
  Return;
End -- Func reduce(j,k)
```

5 Experimental Results

The algorithm described in this paper was implemented within the TLV system [PS96]. TLV is a flexible verification tool implemented at the Weizmann Institute of Science. TLV provides a programming environment which uses BDDs as its basic data type [Bry86]. Deductive and algorithmic verification methods are implemented as procedures written within this environment. We extended TLV's functionality by implementing the algorithms in this paper. We consider two examples. The case of an arbiter and the case of a lift controller.

5.1 Arbiter

We consider the case of an arbiter. Our arbiter has n input lines in which clients request permissions and n output lines in which the clients are granted permission. We assume that initially the requests are set to zero, once a request has been made it cannot be withdrawn, and that the clients are fair, that is once a grant to a certain client has been given it eventually releases the resource by lowering its request line. Formally, the assumption on the environment in LTL format is below.

$$\bigwedge_i (\overline{r_i} \wedge \Box\,((r_i{\neq}g_i) \rightarrow (r_i{=}\bigcirc\,r_i)) \wedge \Box\,((r_i \wedge g_i) \rightarrow \Diamond\,\overline{r_i}))$$

We expect the arbiter to initially give no grants, give at most one grant at a time (mutual exclusion), give only requested grants, maintain a grant as long as it is requested, to satisfy (eventually) every request, and to take grants that are no longer needed. Formally, the requirement from the system in LTL format is below.

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

The resulting game is G: $\langle V, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \varphi\rangle$ where

- $\mathcal{X} = \{r_i \mid i = 1, \ldots, n\}$
- $\mathcal{Y} = \{g_i \mid i = 1, \ldots, n\}$
- $\Theta = \bigwedge_i(\overline{r_i} \wedge \overline{g_i})$
- $\rho_e = \bigwedge_i((r_i{\neq}g_i) \rightarrow (r_i'{=}r_i))$
- $\rho_s = \bigwedge_{i\neq j} \neg(g_i' \wedge g_j') \wedge \bigwedge_i((r_i{=}g_i) \rightarrow (g_i' = g_i))$
- $\varphi = \bigwedge_i \Box((r_i{\wedge}g_i) \rightarrow \Diamond\,\overline{r_i}) \rightarrow \bigwedge_i \Box((r_i{\wedge}\overline{g_i}) \rightarrow \Diamond\,g_i){\wedge}\Box((\overline{r_i}{\wedge}g_i) \rightarrow \Diamond\,\overline{g_i})$

We simplify φ by replacing $\Box((r_i \wedge g_i) \rightarrow \Diamond\,\overline{r_i})$ by $\Box\,\Diamond\,\neg(r_i \wedge g_i)$ and replacing $\Box((r_i \wedge \overline{g_i}) \rightarrow \Diamond\,g_i)$ and $\Box((\overline{r_i} \wedge g_i) \rightarrow \Diamond\,\overline{g_i})$ by $\Box\,\Diamond(r_i{=}g_i)$. The first simplification is allowed because whenever $r_i \wedge g_i$ holds, the next value of g_i is true. The second simplification is allowed because whenever $r_i \wedge \overline{g_i}$ or $\overline{r_i} \wedge g_i$ holds, the next value of r_i is equal to the current. This results with the simpler goal:

$$\varphi = \bigwedge_i \Box\,\Diamond\,\neg(r_i \wedge g_i) \rightarrow \bigwedge_i \Box\,\Diamond(r_i{=}g_i)$$

In Fig. 2, we present graphs of the run time and size of resulting implementations for the Arbiter example. Implementation sizes are measured in number of BDD nodes.

In Fig. 3 we include the explicit representation of the arbiter for two clients resulting from the application of our algorithm.

5.2 Lift Controller

We consider the case of a lift controller. We build a lift controller for n floors. We assume n button sensors. The lift may be requested on every floor, once the lift has been called on some floor the request cannot be withdrawn. Initially, on all floors there are no requests. Once a request has been fulfilled it is removed. Formally, the assumption on the environment in LTL format is below.

$$\bigwedge_i \left(\overline{b_i} \wedge \Box\,\left((b_i \wedge f_i) \rightarrow \bigcirc\,\overline{b_i}\right) \wedge \Box\,((b_i \wedge \neg f_i) \rightarrow \bigcirc\,b_i)\right)$$

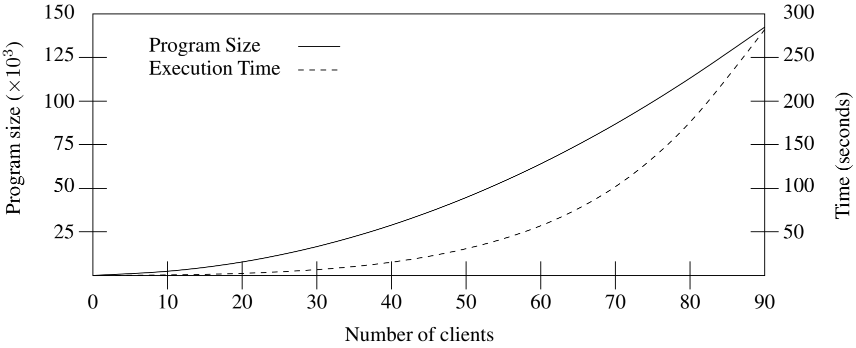

Fig. 2. Running times and program size for the Arbiter example

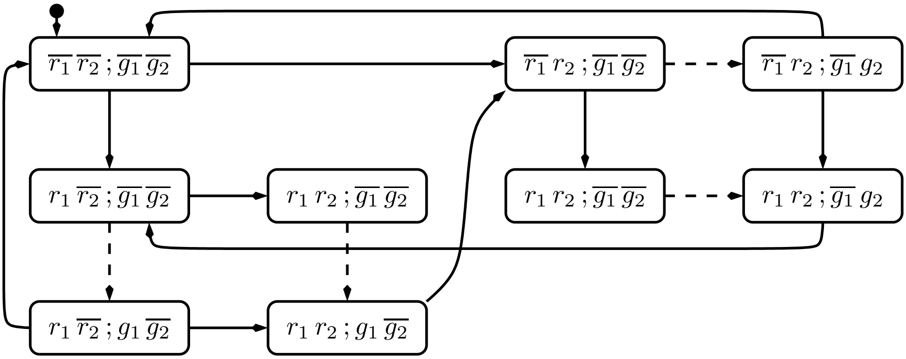
$\overline{r_1}\,\overline{r_2}\,;\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,r_2\,;\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,r_2\,;\overline{g_1}\,g_2$
$r_1\,\overline{r_2}\,;\overline{g_1}\,\overline{g_2}$
$r_1\,r_2\,;\overline{g_1}\,\overline{g_2}$
$r_1\,r_2\,;\overline{g_1}\,\overline{g_2}$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

Fig. 3. Arbiter for 2

We expect the lift to initially start on the first floor. We model the location of the lift by an n bit array. Thus we have to demand mutual exclusion on this array. The lift can move at most one floor at a time, and eventually satisfy every request. Formally, the requirement from the system in LTL format is below.

$$\begin{aligned}&\Box(up \rightarrow sb) \;\wedge\; \Box\Diamond(f_1 \vee sb) \;\wedge\; \textstyle\bigwedge_{i\neq j} \Box\neg(f_i \wedge f_j) \qquad\qquad\qquad \wedge\\ &\textstyle\bigwedge_i\, ((i{=}1 \wedge f_i \vee i{\neq}1 \wedge \neg f_i) \wedge \Box\Diamond(b_i \rightarrow f_i) \wedge \Box(f_i \rightarrow \bigcirc(f_i \vee f_{i-1} \vee f_{i+1})))\end{aligned}$$

where $up = \bigvee_i (f_i \wedge \bigcirc f_{i+1})$ denotes that the lift moves one floor up, and $sb = \bigvee_i b_i$ denotes that at least one button is pressed. The requirement $\Box(up \rightarrow sb)$ states that the lift should not move up unless some button is pressed. The liveness requirement $\Box\Diamond(f_1 \vee sb)$ states that either some button is pressed infinitely many times, or the lift parks at floor f_1 infinitely many times. Together they imply that when there is no active request, the lift should move down and park at floor f_1.

In Fig. 4 we present graphs of the run time and the size of the resulting implementations for different number of floors. As before, implementation sizes are measured in number of BDD nodes.

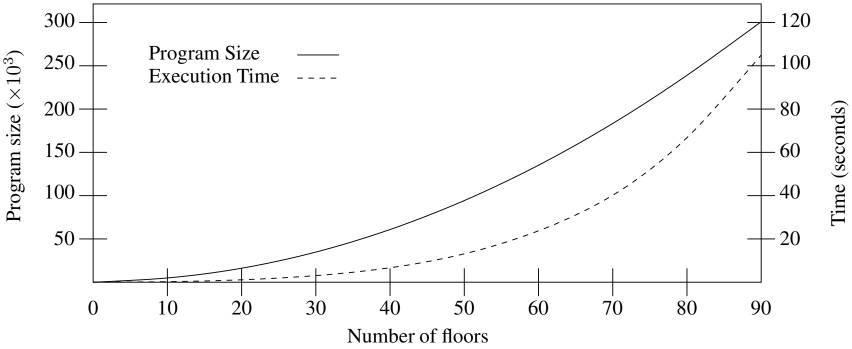

Fig. 4. Running times and program size for the Lift example

6 Extensions

The class of specifications to which the N^3-synthesis algorithm is applicable is wider than the limited form presented in Equation (1). The algorithm can be applied to any specification of the form $(\bigwedge_{i=1}^{m} \varphi_i) \rightarrow (\bigwedge_{j=1}^{n} \psi_j)$, where each φ_i and ψ_j can be specified by an LTL formula of the form $\square \diamondsuit q$ for a past formula q. Equivalently, each φ_i and ψ_j should be specifiable by a deterministic Büchi automaton. This is, for example, the case of the original version of the Arbiter, where the liveness conjuncts were each a response formula of the form $\square (p \rightarrow \diamondsuit q)$.

The way we deal with such a formula is to add to the game additional variables and a transition relation which encodes the deterministic Büchi automaton. For example, to deal with a formula $\square (p \rightarrow \diamondsuit q)$, we add to the game variables a new Boolean variable x with initial condition $x = 1$, and add to the transition relation ρ_e the additional conjunct

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

Table 1. Experiments for Arbiter

N	Recurrence Properties	Response Properties
4	0.05	0.33
6	0.06	0.89
8	0.13	1.77
10	0.25	3.04
12	0.48	4.92
14	0.87	7.30
16	1.16	10.57
18	1.51	15.05
20	1.89	20.70
25	3.03	43.69
30	4.64	88.19
35	6.78	170.50
40	9.50	317.33

We replace in the specification the sub-formula $\square(p \rightarrow \diamond q)$ by the conjunct $\square \diamond x$. It is not difficult to see that this is a sound transformation. That is, the formula $\square(p \rightarrow \diamond q)$ is satisfied by a sequence σ iff there exists an interpretation of the variable x which satisfies the added transition relation and also equals 1 infinitely many times.

Indeed, the in Table 1 we present the performance results of running the Arbiter example with the original specification, to which we applied the above transformation from response to recurrence formulas. The first column presents the results, when the liveness requirements are given as the recurrence formulas $\square \diamond(r_i = g_i)$. In the second column, we present the results for the case that we started with the original requirements $\square(r_i \rightarrow \diamond)g_i$, and then transformed them into recurrence formulas according to the recipe presented above.

7 Conclusions

We presented an algorithm that solves realizability and synthesis for a subset of LTL. For this subset the algorithm works in cubic time. We also presented an algorithm which reduces the number of states in the synthesized module for the case that the specification is stuttering insensitive.

We have shown that the approach can be applied to wide class of formulas, which covers the full set of generalized reactivity[1] properties. We expect both the system and the environment to be realized by hardware designs. Thus, the temporal semantics of both the system and the environment have a specific form and the implication between the two falls in the set of formulas that we handle. Generalized reactivity[1] certainly covers all the specifications we have so far considered in the Prosyd project. We believe that modifications similar to the ones described in Section 6 would be enough to allow coverage of specifications given in languages such as PSL or FORSPEC [AO04, AFF^{+}02].

Acknowledgments

We thank P. Madhusudan for suggesting that enumerating the states of the controller may be very inefficient.

References

[AFF^{+}02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In 8^{th} *TACAS*, LNCS 2280, 2002.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[AO04] Inc. Accellera Organization. Formal semantics of Accellera(c) property specification language. Appendix B of http://www.eda.org/vfv/docs/PSL-v1.1.pdf, January 2004.

[AT04] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25.

[EL86] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional modal μ-calculus. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 267–278, 1986.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. AMS, 1997.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 163:203–243, 2000.

[KPP05] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Comp.*, 200(1):36–61, 2005.

[Lic91] O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, Weizmann Institute of Science, 1991.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 652–671. Springer-Verlag, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symp. Found. of Comp. Sci.*, pages 746–757, 1990.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 184–195, 1996.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Churc's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.