Effective Synthesis of Asynchronous Systems from GR(1) Specifications

Courant Institute of Mathematical Sciences, NYU - Technical Report TR2011-944 (an extended version of a VMCAI'12 paper)*

Uri Klein[1], Nir Piterman[2], and Amir Pnueli[1]

[1] Courant Institute of Mathematical Sciences, New York University
uriklein@courant.nyu.edu
[2] Department of Computer Science, University of Leicester
nir.piterman@leicester.ac.uk

Abstract. We consider automatic synthesis from linear temporal logic specifications for *asynchronous* systems. We aim the produced reactive systems to be used as software in a multi-threaded environment. We extend previous reduction of asynchronous synthesis to synchronous synthesis to the setting of multiple input and multiple output variables. Much like synthesis for synchronous designs, this solution is not practical as it requires determinization of automata on infinite words and solution of complicated games. We follow advances in synthesis of synchronous designs, which restrict the handled specifications but achieve scalability and efficiency. We propose a heuristic that, in some cases, maintains scalability for asynchronous synthesis. Our heuristic can prove that specifications are realizable and extract designs. This is done by a reduction to synchronous synthesis that is inspired by the theoretical reduction.

1 Introduction

One of the most ambitious and challenging problems in reactive systems design is the automatic synthesis of programs from logical specifications. It was suggested by Church [4] and subsequently solved by two techniques [3, 22]. In [18] the problem was set in a modern context of synthesis of reactive systems from Linear Temporal Logic (LTL) specifications. The synthesis algorithm converts a LTL specification to a Büchi automaton, which is then determinized [18]. This double translation may be doubly exponential in the size of φ. Once the deterministic automaton is obtained, it is converted to a Rabin game that can be solved in time $n^{O(k)}$, where n is the number of states of the automaton (double exponential in φ) and k is a measure of topological complexity (exponential in φ). This algorithm is tight as the problem is 2EXPTIME-hard [18].

This unfortunate situation led to extensive research on ways to bypass the complexity of synthesis (e.g., [15, 9, 16]). The work in [16] is of particular interest to us. It achieves scalability by restricting the type of handled specifications. This

* Supported in part by National Science Foundation grant CNS-0720581

led to many applications of synthesis in various fields [1, 2, 6, 12, 27, 29, 28, 13, 14, 7, 8]. So, in some cases, synthesis of designs from their temporal specifications is feasible.

These results relate to the case of synchronous synthesis, where the synthesized system is synchronized with its environment. At every step, the environment generates new inputs and the system senses all of them and computes a response. This is the standard computational model for hardware designs.

Here, we are interested in synthesis of asynchronous systems. Namely, the system may not sense all the changes in its inputs, and its responses may become visible to the external world (including the environment) with an arbitrary delay. Furthermore, the system accesses one variable at a time while in the synchronous model all inputs are observed and all outputs are changed in a single step. The asynchronous model is the most appropriate for representing reactive software systems that communicate via shared variables on a multi-threaded platform.

We illustrate the difference between the two types of synthesis (synchronous and asynchronous) by the following trivial example. Consider a system with a single input x and a single output y, both Booleans. The behavioral specification is given by the temporal formula

$$\varphi_1 : \Box(x \leftrightarrow y)$$

stating that, at all computation steps, it is required that the output should equal the input. Obviously, this specification calls for an implementation of a module that will consistently 'copy' the input to the output. As usual, the synthesis problem is to find a module such that, for all possible sequences of values appearing on input x, will maintain the specification φ_1.

It is not difficult to see that specification φ_1 is synchronously realizable. That is, there exists a synchronous module that maintains the specification φ_1. Such a module can be defined by having the initial condition $\theta : y \leftrightarrow x$ and the transition relation $\rho : y' \leftrightarrow x'$. This presentation is based on the notion of a Fair Discrete System (FDS) as presented, for example, in [16]. (A hardware module implementing this specification would eventually amount to a wire connecting the input to the output.)

On the other hand, the specification φ_1 is not asynchronously realizable. That is, there does not exist an asynchronous module such that, for all possible sequences of x-values, it will maintain the relation $x \leftrightarrow y$. The reason is that if x changes too rapidly, the system cannot observe all these changes and respond quickly enough. In particular, since in an asynchronous setting steps of the environment and of the system interleave, there is no way (unlike in the synchronous model) that x and y can both change in the same step.

In [19], Pnueli and Rosner reduce asynchronous synthesis to synchronous synthesis. Their technique, which we call the Rosner reduction, converts a specification $\varphi(x; y)$ with single input x and single output y to a specification $\mathcal{X}(x, r; y)$. The new specification relates to an additional input r. They show that φ is asynchronously realizable **iff** $\mathcal{X}$ is synchronously realizable and how to translate a synchronous implementation of $\mathcal{X}$ to an asynchronous implementation of φ.

Our first result is an extension of the Rosner reduction to specifications with multiple input and output variables. Pnueli and Rosner assumed that the system alternates between reading its input and writing its output. For multiple variables, we assume cyclic access to variables: first reading all inputs, then writing all outputs (each in a fixed order). We show that this interaction mode is not restrictive as it is equivalent (w.r.t. synthesis) to the model in which the system chooses its next action (whether to read or to write and which variable).

Combined with [18], the reduction from asynchronous to synchronous synthesis presents a complete solution to the multiple-variables asynchronous synthesis problem. Unfortunately, much like in the synchronous case, it is not 'effective'. Furthermore, even if φ is relatively simple (for example, belongs to the class of $GR(1)$ formulae that is handled in [16]), the formula $\mathcal{X}$ is considerably more complex and requires the full treatment of [18].

Consequently, we propose a method to bypass this full reduction. In the invited paper [17] we outlined the principles of an approach to bypass the complexity of asynchronous synthesis. Our approach applied to specifications that relate to one input and one output, both Boolean. We presented heuristics that can be used to prove unrealizability and to prove realizability. The approximation of unrealizability called for the construction of a weakening that could prove unrealizability through a simpler reduction to synchronous synthesis. In this paper we show how to extend this result to multiple variables. Our results are based on an extension of the Rosner reduciton, which we present. In [17] we also outlined an approach to approximate realizability. We suggested to strengthen specifications and an alternative reduction to synchronous synthesis for such strengthened specifications. Here we substantiate these conjectured ideas by completing and correcting the details of that approach and extending it to multiple value variables and multiple outputs. We show that the ideas portrayed in [17] require to even further restrict the type of specifications and a more elaborate reduction to synchronous synthesis (even for the Boolean one-input one-output case of [17]). We show that when the system has access to the 'entire state' of the environment (this is like the environment having one multiple value variable) there are cases where a simpler reduction to synchronous synthesis can be applied. We give a conversion from the synchronous implementation to an asynchronous implementation realizing the original specification.

To our knowledge, this is the first 'easy' case of asynchronous synthesis identified. With connections to partial-information games and synthesis with nondeterministic environments, we find this to be a very important research direction.

This technical report is an extended version of [10].

2 Preliminaries

2.1 Temporal Logic

We describe an extension of Quantified Propositional Temporal Logic (QPTL) [24] with *stuttering quantification*. We refer to this extended logic as QPTL. Let

X be a set of variables ranging over the same finite domain D. The syntax of QPTL is defined according to the following grammar.

$$\tau \ ::= \ x = d, \text{ where } x \in X \text{ and } d \in D$$
$$\varphi \ ::= \ \tau \parallel \neg\varphi \parallel \varphi \vee \varphi \parallel \bigcirc \varphi \parallel \ominus \varphi \parallel \varphi \mathcal{U} \varphi \parallel \varphi \mathcal{S} \varphi \parallel (\exists x).\varphi \parallel (\exists^{\approx} x).\varphi$$

where τ are *atomic formulae* and φ are *QPTL formulae* (formulae, for short).

We use the abbreviations (here, $d \in D$, $x, y \in X$, and ψ, ψ_1, ψ_2 are formulae): $x \neq d$ for $\neg(x = d)$, T for $x = d \vee x \neq d$, F for $\neg\mathrm{T}$, $\psi_1 \wedge \psi_2$ for $\neg(\neg\psi_1 \vee \neg\psi_2)$, $\psi_1 \rightarrow \psi_2$ for $\neg\psi_1 \vee \psi_2$, $\psi_1 \leftrightarrow \psi_2$ for $(\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1)$, $(\forall x).\psi$ for $\neg(\exists x).(\neg\psi)$, $(\forall^{\approx} x).\psi$ for $\neg(\exists^{\approx} x).(\neg\psi)$, $\diamondsuit \psi$ for $\mathrm{T}\,\mathcal{U}\psi$, $\square \psi$ for $\neg \diamondsuit \neg\psi$, $\diamondminus \psi$ for $\mathrm{T}\,\mathcal{S}\,\psi$, $\boxminus \psi$ for $\neg \diamondminus \neg\psi$, $\psi_1 \mathcal{W} \psi_2$ for $\psi_1 \mathcal{U} \psi_2 \vee \square \psi_1$, $\psi_1 \mathcal{B}\, \psi_2$ for $\psi_1 \mathcal{S}\, \psi_2 \vee \boxminus \psi_1$, $x = y$ for $\bigvee_{d \in D}(x = d \wedge y = d)$, $x \neq y$ for $\neg(x = y)$, $x = \ominus y$ for $\bigvee_{d \in D}(x = d \wedge \ominus y = d)$, $\owns \psi$ for $\neg \ominus \neg\psi$, and $\psi_1 \Rightarrow\!\!\!\!\succ \psi_2$ for $\square(\psi_1 \rightarrow \psi_2)$. For a set $\hat{X} = \{x_1, \ldots, x_k\}$ of variables, where $\hat{X} \subseteq X$, we write $(\exists \hat{X}).\psi$ for $(\exists x_1) \cdots (\exists x_k).\psi$ and similarly for $(\forall \hat{X}).\psi$. We sometimes list variables and sets, e.g., $(\exists \hat{X}, y).\psi$ instead of $(\exists \hat{X} \cup \{y\}).\psi$. Also, for a Boolean variable r we write r for $r = 1$ and $\overline{r}$ for $r = 0$.

LTL does not allow the $\exists$ and $\exists^{\approx}$ operators. We stress that a formula φ is written over the variables in a set X by writing $\varphi(X)$. If variables are partitioned to *inputs* X and *outputs* Y, we write $\varphi(X; Y)$. We call such formulae *specifications*. We sometimes list the variables in X and Y, e.g., $\varphi(x_1, x_2; y)$.

The semantics of QPTL is given with respect to computations and locations in them. A *computation* σ is an infinite sequence $a_0, a_1, \ldots$, where for every $i \geq 0$ we have $a_i \in D^X$. That is, a computation is an infinite sequence of value assignments to the variables in X. We denote by $\sigma[i, j]$ the subsequence $a_i, \ldots, a_j$, where j could be ∞. For an assignment $a \in D^X$ and a variable $x \in X$ we write $a[x]$ for the value assigned to x by a. If $X = \{x_1, \ldots, x_n\}$, we freely use the notation $(a_{i_1}[x_1], \ldots, a_{i_n}[x_n])$ for the assignment a such that $a[x_j] = a_{i_j}[x_j]$. A computation $\sigma' = a'_0, a'_1, \ldots$ is an *x-variant* of computation $\sigma = a_0, a_1, \ldots$ if for every $i \geq 0$ and every $y \neq x$ we have $a_i[y] = a'_i[y]$. The computation $squeeze(\sigma)$ is obtained from σ as follows. If for all $i \geq 0$ we have $a_i = a_0$, then $squeeze(\sigma) = \sigma$. Otherwise, if $a_0 \neq a_1$ then $squeeze(\sigma) = a_0, squeeze(a_1, a_2, \ldots)$. Finally, if $a_0 = a_1$ then $squeeze(\sigma) = squeeze(a_1, a_2, \ldots)$. That is, by removing repeating assignments, *squeeze* returns a computation in which every two adjacent assignments are different unless the computation ends in an infinite suffix of one assignment. A computation σ' is a *stuttering variant* of σ if $squeeze(\sigma) = squeeze(\sigma')$. This is generalized to finite computations in the natural way.

Satisfaction of a QPTL formula φ over computation $\sigma = a_0, a_1, \ldots$ in location $i \geq 0$, denoted $\sigma, i \models \varphi$, is defined as follows:

1. For an atomic formula $x = d$, we have $\sigma, i \models x = d$ **iff** $a_i[x] = d$.
2. $\sigma, i \models \neg\varphi$ **iff** $\sigma, i \not\models \varphi$.
3. $\sigma, i \models \varphi \vee \psi$ **iff** $\langle \sigma, i \models \varphi$ or $\sigma, i \models \psi$.
4. $\sigma, i \models \bigcirc \varphi$ **iff** $\sigma, i + 1 \models \varphi$.
5. $\sigma, i \models \ominus \varphi$ **iff** $i > 0$ and $\sigma, i - 1 \models \varphi$.
6. $\sigma, i \models \varphi \mathcal{U} \psi$ **iff** for some $j \geq i$, $\sigma, j \models \psi$, and for all k, $i \leq k < j$, $\sigma, k \models \varphi$.

7. $\sigma, i \models \varphi \,\mathcal{S}\, \psi$ **iff** for some $j \leq i$, $\sigma, j \models \psi$, and for all k, $i \geq k > j$, $\sigma, k \models \varphi$.
8. We have $\sigma, i \models (\exists x).\varphi$ **iff** there exists an x-variant σ' of σ such that $\sigma', i \models \varphi$.
9. We have $\sigma, i \models (\exists^{\approx} x).\varphi$ **iff** there exist σ', σ'' and σ''' such that σ' is a stuttering variant of $\sigma[0, i-1]$, σ'' is a stuttering variant of $\sigma[i, \infty]$, σ''' is an x-variant of $\sigma' \cdot \sigma''$, and $\sigma''', |\sigma'| \models \varphi$.

We say that the computation σ satisfies the formula φ, **iff** $\sigma, 0 \models \varphi$.

2.2 Realizability of Temporal Specifications

We define synchronous and asynchronous programs. While the programs themselves are not very different the definition of interaction of a program makes the distinction clear.

Let X and Y be the sets of *inputs* and *outputs*. We stress the different roles of the system and the environment by specializing computations to *interactions*. In an interaction we treat each assignment to $X \cup Y$ as different assignments to X and Y. Thus, instead of using $c \in D^{X \cup Y}$, we use a pair (a, b), where $a \in D^X$ and $b \in D^Y$. Formally, an interaction is $\sigma = (a_0, b_0), (a_1, b_1), \ldots \in (D^X \times D^Y)^\omega$.

A *synchronous program* P_s from X to Y is a function $P_s : (D^X)^+ \mapsto D^Y$. In every step of the computation (including the initial one) the program reads its inputs and updates the values of all outputs (based on the entire history). An interaction σ is called *synchronous interaction* of P if, at each step of the interaction $i = 0, 1, \ldots$, the program outputs (assigns to Y) the value $P_s(a_0, a_1, \ldots, a_i)$, i.e., $b_i = P_s(a_0, \ldots, a_i)$. In such interactions both the environment, which updates input values, and the system, which updates output values, 'act' at each step (where the system responds in each step to an environment action).

A synchronous program is *finite state* if it can be *induced* by a *Labeled Transition System (LTS)*. A LTS is $T = \langle S, I, R, X, Y, L \rangle$, where S is a finite set of *states*, $I \subseteq S$ is a set of *initial states*, $R \subseteq S \times S$ is a *transition relation*, X and Y are disjoint sets of *input* and *output* variables, respectively, and $L : S \mapsto D^{X \cup Y}$ is a *labeling* function. For a state $s \in S$ and for $Z \subseteq X \cup Y$, we define $L(s)|_Z$ to be the restriction of $L(s)$ to the variables of Z. The LTS has to be *receptive*, i.e., be able to accept all inputs. Formally, for every $a \in D^X$ there is some $s_0 \in I$ such that $L(s_0)|_X = a$. For every $s \in S$ and $a \in D^X$ there is some $s_a \in S$ such that $R(s, s_a)$ and $L(s_a)|_X = a$. The LTS T is *deterministic* if for every $a \in D^X$ there is a unique $s_0 \in I$ such that $L(s_0)|_X = a$ and for every $s \in S$ and every $a \in D^X$ there is a unique $s_a \in S$ such that $R(s, s_a)$ and $L(s_a)|_X = a$. Otherwise, it is *nondeterministic*. A deterministic LTS T induces the synchronous program $P_T : (D^X)^+ \mapsto D^Y$ as follows. For every $a \in D^X$ let $T(a)$ be the unique state $s_0 \in I$ such that $L(s_0)|_X = a$. For every $n > 1$ and $a_1 \ldots a_n \in (D^X)^+$ let $T(a_1, \ldots, a_n)$ be the unique $s \in S$ such that $R(T(a_1, \ldots, a_{n-1}), s)$ and $L(s)|_X = a_n$. For every $a_1 \ldots a_n \in (D^X)^+$ let $P_T(a_1, \ldots, a_n)$ be the unique $b \in D^Y$ such that $b = L(T(a_1, \ldots, a_n))|_Y$. We note that nondeterministic LTS do not induce programs. As nondeterministic LTS can always be pruned to deterministic LTS, we find it acceptable to produce nondeterministic LTS as a representation of a set of possible programs.

An *asynchronous program* P_a from X to Y is a function $P_a : (D^X)^* \mapsto D^Y$. Note that the first value to outputs is set before seeing inputs. As before, the program receives all inputs and updates all outputs. However, the definition of an interaction takes into account that this may not happen instantaneously.

A *schedule* is a pair (R, W) of sequences $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$ of *reading points* and *writing points* such that $r_1^1 > 0$ and for every $i > 0$ we have $r_i^1 < r_i^2 < \cdots < r_i^n < w_i^1$ and $w_i^1 < w_i^2 < \cdots < w_i^m < r_{i+1}^1$. It identifies the points where each of the input variables is read and the points where each of the output variables is written. The order establishes that reading and writing points occur cyclically. When the distinction is not important, we call reading points and writing points *$I\backslash O$-points*.

An interaction is called *asynchronous interaction* of P_a for (R, W) if $b_0 = P_a(\epsilon)$, and for every $i > 0$, every $j \in \{1, \ldots, m\}$, and every $w_i^j \leq k < w_{i+1}^j$:

$$b_k[j] = P_a((a_{r_1^1}[1], \ldots, a_{r_1^n}[n]), (a_{r_2^1}[1], \ldots, a_{r_2^n}[n]), \ldots, (a_{r_i^1}[1], \ldots, a_{r_i^n}[n]))[j].$$

Also, for every $j \in \{1, \ldots, m\}$ and every $0 < k < w_1^j$, we have that $b_k[j] = b_0[j]$.

In asynchronous interactions, the environment may update the input values at each step. However, the system is only aware of the values of inputs at reading points and responds by outputting the appropriate variables at writing points. In particular, the system is not even aware of the amount of time that passes between the two adjacent time points (read-read, read-write, or write-read). That is, output values depend only on the values of inputs in earlier reading points.

An asynchronous program is *finite state* if it can be *asynchronously induced* by an *Initialized LTS (ILTS)*. An ILTS is $T = \langle T_s, i \rangle$, where $T_s = \langle S, I, R, X, Y, L \rangle$ is a LTS, and $i \in D^Y$ is an *initial assignment*. We sometimes abuse notations and write $T = \langle S, I, R, X, Y, L, i \rangle$. Determinism is defined just as for LTS. Similarly, given $a_1, \ldots, a_n \in (D^X)^+$ we define $T(a_1, \ldots, a_n)$ as before. A deterministic ILTS T asynchronously induces the program $P_T : (D^X)^* \mapsto D^Y$ as follows. Let $P_T(\epsilon) = i$ and for every $a_1 \ldots a_n \in (D^X)^+$ we have $P_T(a_1, \ldots, a_n)$ as before. As i is a unique initial assignment, we force ILTS to induce only asynchronous programs that deterministically assign a single initial value to outputs. All our results work also with a definition that allows nondeterministic choice of initial output values (that do not depend on the unavailable inputs).

Definition 1 (synchronous realizability). *A* LTL *specification $\varphi(X; Y)$ is* **synchronously realizable** *if there exists a synchronous program P_s such that all synchronous interactions of P_s satisfy $\varphi(X; Y)$. Such a program P_s is said to* synchronously realize *$\varphi(X; Y)$.* Synchronous realizability *is often simply shortened to* realizability. **Asynchronous realizability** *is defined similarly with asynchronous programs and all asynchronous interactions for all schedules.*

Synthesis is the process of automatically constructing a program P that (synchronously/asynchronously) realizes a specification $\varphi(X; Y)$. We freely write that a LTS realizes a specification in case that the induced program satisfies it.

The following theorem is proven in [18].

Theorem 1 ([18]). *Deciding whether a specification $\varphi(X;Y)$ is synchronously realizable is 2EXPTIME-complete. Furthermore, if $\varphi(X;Y)$ is synchronously realizable the same decision procedure can extract a LTS that realizes $\varphi(X;Y)$.*

2.3 Normal Form of Specifications

We give a normal form of specifications describing an interplay between a *system* s and an *environment* e. Let X and Y be disjoint sets of input and output variables, respectively. For $\alpha \in \{e, s\}$, the formula $\varphi_\alpha(X;Y)$, which defines the allowed actions of α, is a conjunction of:

1. I_α (*initial condition*) – a Boolean formula (equally, an assertion) over $X \cup Y$, describing the initial state of α. The formula I_s may refer to all variables and I_e may refer only to the variables X.
2. $\square\, S_\alpha$ (*safety component*) – a formula describing the transition relation of α, where S_α describes the update of the locally controlled state variables (identified by being *primed*, e.g., x' for $x \in X$) as related to the current state (unprimed, e.g., x), except that s can observe X's next values.
3. L_α (*liveness component*) – each L_α is a conjunction of $\square \diamondsuit p$ formulae where p is a Boolean formula.

In the case that a specification includes temporal past formulae instead of the Boolean formulae in any of the three conjuncts mentioned above, we assume that a pre-processing of the specification was done to translate it into another one that has the same structure but without the use of past formulae. This can be always achieved through the introduction of fresh Boolean variables that implement temporal testers for past formulae [21]. Therefore, without loss of generality, we discuss in this work only such past-formulae-free specifications.

We abuse notations and write φ_α also as a triplet $\langle I_\alpha, S_\alpha, L_\alpha \rangle$.

Consider a pair of formulae $\varphi_\alpha(X;Y)$, for $\alpha \in \{e, s\}$ as above. We define the specification $Imp(\varphi_e, \varphi_s)$ to be $(I_e \wedge \square\, S_e \wedge L_e) \rightarrow (I_s \wedge \square\, S_s \wedge L_s)$. For such specifications, the *winning condition* is the formula $L_e \rightarrow L_s$, which we call $GR(1)$. Synchronous synthesis of such specifications was considered in [16].

2.4 The Rosner Reduction

In [19], Pnueli and Rosner show how to use synchronous realizability to solve asynchronous realizability. They define, what we call, *the Rosner reduction*. It translates a specification $\varphi(X;Y)$, where $X = \{x\}$ and $Y = \{y\}$ are singletons, into a specification $\mathcal{X}(x, r; y)$ that has an additional Boolean input variable r. The new variable r is called the *Boolean scheduling variable*. Intuitively, the Boolean scheduling variable defines all possible schedules for one-input one-output systems . When it changes from zero to one it signals a reading point and when it changes from one to zero it signals a writing point. Given specification $\varphi(X;Y)$, we define the *kernel formula* $\mathcal{X}(x, r; y)$:

$$\underbrace{\overline{r} \wedge \Box \diamondsuit r \wedge \Box \diamondsuit \overline{r}}_{\alpha(r)} \rightarrow \underbrace{\begin{pmatrix} \varphi(x;y) & \wedge \\ (r \vee \ominus \overline{r}) \Rightarrow (y = \ominus y) & \wedge \\ (\forall^{\approx} \widetilde{x}).[(r \wedge \ominus \overline{r}) \Rightarrow (x = \widetilde{x})] \rightarrow \varphi(\widetilde{x};y) & \end{pmatrix}}_{\beta(x,r;y)}$$

According to $\alpha(r)$, the first $I\backslash O$-point, where r changes from zero to one, is a reading point and there are infinitely many reading and writing points. Then, $\beta(x,r;y)$ includes three parts: (a) the original formula $\varphi(x;y)$ must hold, (b) outputs obey the scheduling variable, i.e., in all points that are not writing points the value of y does not change, and (c) if we replace all the inputs except in reading points, then the same output still satisfies the original formula [1].

The following theorem is proven in [19].

Theorem 2 ([19]). *The specification $\varphi(x;y)$ is asynchronously realizable* **iff** *the specification $\mathcal{X}(x,r;y)$ is synchronously realizable. Given a program that synchronously realizes $\mathcal{X}(x,r;y)$ it can be converted in linear time to a program asynchronously realizing $\varphi(x;y)$.*

Pnueli and Rosner also show how the standard techniques for realizability of LTL [18] can handle stuttering quantification of the form appearing in $\mathcal{X}(x,r;y)$.

3 Expanding the Rosner Reduction to Multiple Variables

In this section we describe an expansion of the Rosner reduction to handle specifications with multiple input and output variables. The reduction reduces asynchronous synthesis to synchronous synthesis. Without loss of generality, fix a LTL specification $\varphi(X;Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$.

We propose the *generalized Rosner reduction*, which translates $\varphi(X;Y)$ into $\mathcal{X}^{n,m}(X \cup \{r\};Y)$. The specification uses an additional input variable r, called the *scheduling variable*, which ranges over $\{1, \ldots, (n+m)\}$ and defines all reading and writing points. Variable x_i may be read by the system whenever r changes its value to i. Variable y_i may be modified whenever r changes to $n+i$. Initially, $r = n+m$ and it is incremented cyclically by 1 (hence, in the first $I\backslash O$-point x_1 is read). Let $i \oplus_k 1$ denote $(i \bmod k) + 1$.

We also denote $[r = (n+i)] \wedge \ominus[r \neq (n+i)]$ by $\mathit{write}_n(i)$ to indicate a state that is a writing point for y_i, $(r = i) \wedge \ominus(r \neq i)$ by $\mathit{read}(i)$ to indicate a state that is a reading point for x_i, $\bigwedge_{d \in D}[(z = d) \leftrightarrow \ominus(z = d)]$ by $\mathit{unchanged}(z)$ to indicate a state where z did not change its value, and $\neg \ominus \mathrm{T}$ by *first* to indicate a state that is the first one in the computation.

The kernel formula $\mathcal{X}^{n,m}(X \cup \{r\};Y)$ is $\alpha^{n,m}(r) \rightarrow \beta^{n,m}(X \cup \{r\};Y)$, where

[1] The first conjunct of $\beta(x,r;y)$, $\varphi(x;y)$, is redundant. It is a consequence of the third conjunct which guarantees that $\varphi(\tilde{x};y)$ is satisfied for a set of sequences of assignments to $\tilde{x}$ which includes the single sequence of assignments to x. We leave this conjunct here, as well as in similar reductions later in this paper, for clarity.

$$\alpha^{n,m}(r) = \left(\begin{array}{l} r = (n+m) \\ \bigwedge_{i=1}^{n+m} \Big[(r=i) \mathrel{\Rightarrow\!\!\!\!\succ} \Big[(r=i) \,\mathcal{U}\, [r = (i \oplus_{n+m} 1)] \Big] \Big] \end{array} \right) \wedge$$

$$\beta^{n,m}(X \cup \{r\}; Y) = \left(\begin{array}{l} \varphi(X;Y) \\ \bigwedge_{i=1}^{m} \Big[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \mathrel{\Rightarrow\!\!\!\!\succ} \mathit{unchanged}(y_i) \Big] \\ (\forall^{\approx} \tilde{X}). \Big[\bigwedge_{i=1}^{n} [\mathit{read}(i) \mathrel{\Rightarrow\!\!\!\!\succ} (x_i = \tilde{x_i})] \Big] \rightarrow \varphi(\tilde{X}; Y) \end{array} \right) \wedge \wedge .$$

There is a 1-1 correspondence between sequences of assignments to r and schedules (R, W). As r is an input variable, the program has to handle all possible assignments to it. This implies that the program handles all possible schedules.

Theorem 3. *The specification $\varphi(X;Y)$ ($|X| = n$, and $|Y| = m$) is asynchronously realizable* **iff** *$\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable. Furthermore, given a program synchronously realizing $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ it can be converted in linear time to a program asynchronously realizing $\varphi(X;Y)$.*

Proof (Sketch): Suppose we have a synchronous program realizing $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ and we want an asynchronous program realizing $\varphi(X;Y)$. An input to the asynchronous program is stretched in order to be fed to the synchronous program. Essentially, every new input to the asynchronous program is stretched so that one variable changes at a time and in addition the new valuation of all input variables is repeated enough times to allow the synchronous program to update all the output variables. This is forced to happen immediately by increasing the scheduling variable r (cyclically) in every input for the synchronous program. This forces the synchronous program to update all output variables and this is the value we use for the asynchronous program. Then, the stuttering quantification over the synchronous interaction shows that an asynchronous interaction that matches these outputs does in fact satisfy $\varphi(X;Y)$.

In the other direction we have an asynchronous program realizing $\varphi(X;Y)$ and have to construct a synchronous program realizing $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. The reply of the synchronous program to every input in which the scheduling variables behaves other than increasing (cyclically) is set to be arbitrary. For inputs where the scheduling variable behaves properly, we can contract the inputs to the reading points indicated by r and feed the resulting input sequence to the asynchronous program. We then change the output variables one by one as indicated by r according to the output of the asynchronous program. In order to see that the resulting synchronous program satisfies $\mathcal{X}$, we note that the stuttering quantification relates precisely to all the possible asynchronous interactions. □

Proof: For clarity, we define $D_r = \{1, \ldots, (n+m)\}$ (the domain of the scheduling variable r). We also use the notation $r_{init} = n + m$ for the 'correct' initial value of r. We shall prove both directions constructively, by reducing each type of program to the other:

$\Leftarrow$ Let $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ be synchronously realized by the synchronous program $P_s : (D^X \times D_r)^+ \mapsto D^Y$. We define the asynchronous program $P_a : (D^X)^* \mapsto D^Y$ as shown below.

$P_a(\epsilon) = P_s((a_{rand}, r_{init}))$, where a_{rand} is some arbitrary assignment to the inputs X. We define the function $dup_{n,m} : (D^X)^+ \mapsto (D^X)^+$ inductively: For all $a \in D^X$, $dup_{n,m}(a) = a^1, \ldots, a^n, \underbrace{a^n, \ldots, a^n}_{m \text{ times}}$ where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ we have $a^i[j] = a[j]$ and for all j such that $i < j \leq n$ we have $a^i[j] = a_{rand}[j]$. For all $k > 1$ and $a_1, \ldots, a_k \in (D^X)^k$, $dup_{n,m}(a_1, \ldots, a_k) = dup_{n,m}(a_1, \ldots, a_{k-1}), a'^1, \ldots, a'^n, \underbrace{a'^n, \ldots, a'^n}_{m \text{ times}}$ where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ we have $a'^i[j] = a_k[j]$ and for all j such that $i < j \leq n$ we have $a'^i[j] = a_{k-1}[j]$. For all $k > 0$ let $r^k = r_1, \ldots, r_k$, where $r_1 = 1$ ($r_1 = (r_{init} \oplus_{n+m} 1)$), and where for all $k \geq i > 1$, $r_i = (r_{i-1} \oplus_{n+m} 1)$. For all $k > 0$, if $a_1, \ldots, a_k \in (D^X)^k$ we define $dupr_{n,m}(a_1, \ldots, a_k) \in (D^X \times D_r)^{(n+m)\cdot k}$ where the projection of $dupr_{n,m}(a_1, \ldots, a_k)$ on the inputs X is $dup_{n,m}(a_1, \ldots, a_k)$, and the projection of $dupr_{n,m}(a_1, \ldots, a_k)$ on the scheduling variable r is $r^{(n+m)\cdot k}$. Finally, for all $k > 0$ such that $a_1, \ldots, a_k \in (D^X)^k$ we define $P_a(a_1, \ldots, a_k) = P_s((a_{rand}, r_{init}), dupr_{n,m}(a_1, \ldots, a_k))$.

We now show that all asynchronous interactions of P_a, for all schedules, satisfy $\varphi(X; Y)$, implying that $\varphi(X; Y)$ is asynchronously realized by P_a. Let (R, W) be a schedule, and let $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be an asynchronous interaction of P_a for this schedule. Denote $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$.

We abuse notation and define the function $dup_{n,m} : (D^X \times D^Y)^+ \mapsto (D^X \times D^Y)^+$ inductively: For all $(a, b) \in D^X \times D^Y$,

$$dup_{n,m}((a, b)) = (a^1, b_0), \ldots, (a^n, b_0), (a^n, b^1), \ldots, (a^n, b^m)$$

where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ $a^i[j] = a[j]$ and for all j such that $i < j \leq n$ $a^i[j] = a_{rand}[j]$. Also, for all $0 < i \leq m$, for all j such that $0 < j \leq i$ $b^i[j] = b[j]$ and for all j such that $i < j \leq m$ $b^i[j] = b_0[j]$. For all $k > 1$ and $(c_1, d_1), \ldots, (c_k, d_k) \in (D^X \times D^Y)^k$,

$$dup_{n,m}((c_1, d_1), \ldots, (c_k, d_k)) = dup_{n,m}\Big((c_1, d_1), \ldots, (c_{k-1}, d_{k-1})\Big), (a^1, d_{k-1}), \ldots, (a^n, d_{k-1}), (a^n, b^1), \ldots, (a^n, b^m)$$

where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ we have $a^i[j] = c_k[j]$ and for all j such that $i < j \leq n$ we have $a^i[j] = c_{k-1}[j]$. Also, for all $0 < i \leq m$, for all j such that $0 < j \leq i$ we have $b^i[j] = d_k[j]$ and for all j such that $i < j \leq m$ we have $b^i[j] = d_{k-1}[j]$. We also abuse notation by defining that for all $k > 0$, if $(c_1, d_1), \ldots, (c_k, d_k) \in (D^X \times D^Y)^k$ then $dupr_{n,m}((c_1, d_1), \ldots, (c_k, d_k)) \in (D^X \times D_r \times D^Y)^{(n+m)\cdot k}$ where the projection of $dupr_{n,m}((c_1, d_1), \ldots, (c_k, d_k))$ on the inputs and outputs $X \cup Y$ is

$dup_{n,m}((c_1,d_1),\ldots,(c_k,d_k))$, and the projection of $dupr_{n,m}((c_1,d_1),\ldots,(c_k,d_k))$ on the scheduling variable r is $r^{(n+m)\cdot k}$. We also apply *dup* and *dupr* to infinite computations. In that case, the result is the limit of the application of the function on all prefixes of the infinite computation.
Consider the computation

$$\sigma' = \Big((a_{r_1^1}[1],\ldots,a_{r_1^n}[n]),(b_{w_1^1}[1],\ldots,b_{w_1^m}[m])\Big),\\ \Big((a_{r_2^1}[1],\ldots,a_{r_2^n}[n]),(b_{w_2^1}[1],\ldots,b_{w_2^m}[m])\Big),\ldots$$

obtained from σ by restricting attention to the values of variables to the appropriate reading and writing points. By construction, we know that the computation $\sigma'' = (a_{rand}, r_{init}, b_0), dupr_{n,m}(\sigma')$ is a synchronous interaction of P_s. Therefore, $\sigma'', 0 \models \mathcal{X}^{n,m}$. Since $\sigma'', 0 \models \alpha^{n,m}$ (due to the way $dupr_{n,m}$ modifies the scheduling variable), we also get that

$$\sigma'', 0 \models (\forall^{\approx}\tilde{X}).\bigwedge_{i=1}^{n}[read(i) \twoheadrightarrow (x_i = \tilde{x_i})] \rightarrow \varphi(\tilde{X};Y). \tag{1}$$

We now define the computation σ''', which is obtained from σ'' by 'stretching' it so that all reading points in σ''' match exactly with the indices of R, and all writing points in it match exactly with the indices of W. By 'matching exactly' we mean that there are no $I \backslash O$ points in σ''' beyond those indicated by the schedule (R, W). When we stretch σ'', the newly added states are copies of their predecessor. That is, if s, t is part of the sequence that needs to be stretched then $s, s, \ldots, s, t$ is the new sequence. As a result, the first state in σ''' is $(a_{rand}, r_{init}, b_0)$, the first state in σ'', and there are exactly r_1^1 copies of it at the prefix of σ'''. The second state of σ'' is duplicated to $r_1^2 - r_1^1$ copies in σ''', the n-th state is duplicated to $r_1^n - r_1^{n-1}$ copies, the $n+1$-th state is duplicated to $w_1^1 - r_1^n$ copies, the $n+m$-th state is duplicated to $w_1^m - w_1^{m-1}$ copies, the $n+m+1$-th state is duplicated to $r_2^1 - w_1^m$ copies, and so on.
From Formula 1, every stuttering variant of σ'' that assigns to $\tilde{X}$ values that agree with X in all reading points, satisfies $\varphi(\tilde{X};Y)$. This is exactly the case of σ. Indeed, σ''' is a stuttering variant of σ'' and agrees with σ on assignments to the outputs Y and to r. It follows that if we consider the assignment of σ to X as an assignment to $\tilde{X}$ added to σ''', we get the required result that $\sigma, 0 \models \varphi(X,Y)$ which concludes the proof of this direction.

$\Rightarrow$ Let $\varphi(X;Y)$ be asynchronously realized by the asynchronous program $P_a : (D^X)^* \mapsto D^Y$. We define the synchronous program $P_s : (D^X \times D_r)^+ \mapsto D^Y$ as shown below.
Given a sequence $a_1,\ldots,a_k \in (D^X \times D_r)^k$, we denote $r_i = a_i[r]$. For all $k > 0$ and $a_1,\ldots,a_k \in (D^X \times D_r)^k$, if $r_1 \neq r_{init}$ or there exists some index $1 < j \leq n$ such that $(r_j \neq r_{j-1}) \wedge (r_j \neq (r_{j-1} \oplus_{n+m} 1))$, then $P_s(a_1,\ldots,a_k,\ldots) = b_{rand}$ (for all prefixes of computations with the prefix $a_1,\ldots,a_k$), where b_{rand} is some arbitrary assignment to the outputs Y. From this point onwards, we

handle only elements of $(D^X \times D_r)^k$ for which the above condition does not hold and which are, therefore, 'compliant' with the initial condition and transition relation implied by $\alpha^{n,m}(r)$. It is worthwhile to note that monitoring this condition could be done 'on-line' while P_s gets more and more inputs, without any increase in complexity.

For all $k > 0$ and given $a_1, \ldots, a_k \in (D^X \times D_r)^k$, we define the *implied prefixed schedule* for $a_1, \ldots, a_k$, (R^k, W^k), to be a prefix of some schedule. We note that all possible extensions of $a_1, \ldots, a_k$ that satisfy $\alpha^{n,m}$ agree on the prefix (R^k, W^k) of their implied schedules. For (R^k, W^k) to be the (unique) prefixed schedule implied by $a_1, \ldots, a_k$, we require that $|R^k| + |W^k|$ equals the number of times r changes its value in $a_1, \ldots, a_k$. Therefore, (R^k, W^k) represents exactly all the $I\backslash O$ points of $a_1, \ldots, a_k$. For all $k > 1$ and $a_1, \ldots, a_k \in (D^X \times D_r)^k$, if (R^k, W^k) is the prefixed schedule implied by $a_1, \ldots, a_k$, let $R^k = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots, r_t^1, \ldots, r_t^s$ and $W^k = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots, w_p^1, \ldots, w_p^q$. Let c_i (for all i) be the projection of a_i on the inputs X. We define $b_{prev(s,t)} \in D^Y$ to be the value assigned by P_a to the outputs at the end of the previous-to-most-recent 'full $I\backslash O$ cycle':

$$b_{prev(s,t)} = P_a((c_{r_1^1}[1], \ldots, c_{r_1^n}[n]), (c_{r_2^1}[1], \ldots, c_{r_2^n}[n]), \ldots, (c_{r_g^1}[1], \ldots, c_{r_g^n}[n]))$$

where if $s = n$ then $g = t - 1$ and otherwise $g = t - 2$. We also define $b_{last(s,t)} \in D^Y$ to be the value assigned by P_a to the outputs at the end of the most recent 'full $I\backslash O$ cycle':

$$b_{last(s,t)} = P_a((c_{r_1^1}[1], \ldots, c_{r_1^n}[n]), (c_{r_2^1}[1], \ldots, c_{r_2^n}[n]), \ldots, (c_{r_f^1}[1], \ldots, c_{r_f^n}[n]))$$

where if $s = n$ then $f = t$ and otherwise $f = t - 1$. Note that in some cases (when t is 'too small'), $b_{prev(s,t)} = P_a(\epsilon)$ or $b_{last(s,t)} = P_a(\epsilon)$. The output of P_s, $P_s(a_1, \ldots, a_k)$, should always be some combination of $b_{last(s,t)}$ and $b_{prev(s,t)}$, based on the output variables of Y that were already updated in the most recent 'writing cycle' as indicated by w_p^q. Hence, $P_s(a_1, \ldots, a_k) = b_{real(s,t,q)}$ where for all $0 < i \leq m$, if $i > q$ then $b_{real(s,t,q)}[i] = b_{prev(s,t)}[i]$, and otherwise $b_{real(s,t,q)}[i] = b_{last(s,t)}[i]$. Note that as a result of this definition of P_s, as long as $a_1, \ldots, a_k$ contains no $I\backslash O$ points, $P_s(a_1, \ldots, a_k) = P_a(\epsilon)$. Particularly, for all $a \in D^X \times D_r$, $P_s(a) = P_a(\epsilon)$.

We now show that all synchronous interactions of P_s satisfy $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, implying that $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realized by P_s. Let $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be a synchronous interaction of P_s.

If $\sigma, 0 \not\models \alpha^{n,m}(r)$, then trivially $\sigma, 0 \models \mathcal{X}^{n,m}$ and we are done. Otherwise, we observe that all computations σ' that are $\tilde{X}$- variants of stuttering variants of σ, in which X and $\tilde{X}$ agree in all reading points, are asynchronous interactions of P_a for the schedule implied by the values of r in σ'. Hence, by correctness of P_a, for every such σ' it holds that $\sigma', 0 \models \varphi(\tilde{X}; Y)$. It follows that $\sigma, 0 \models (\forall^{\approx}\tilde{X}).\bigwedge_{i=1}^{n}[\mathit{read}(i) \twoheadrightarrow (x_i = \tilde{x_i})] \rightarrow \varphi(\tilde{X}; Y)$ and, particularly, also that $\sigma, 0 \models \varphi(X; Y)$. By construction, $\sigma, 0 \models \bigwedge_{i=1}^{m}[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \twoheadrightarrow \mathit{unchanged}(y_i)$ and we finally conclude (given that σ satisfies $\alpha^{n,m}(r)$

as well as all of the three conjuncts on the right-hand-side of $\mathcal{X}^{n,m}$), that $\sigma, 0 \models \mathcal{X}^{n,m}(X \cup \{r\}; Y)$. This concludes the proof.

□

In principle, this theorem provides a complete solution to the problem of asynchronous synthesis (with multiple inputs and outputs). Implementing this solution, however, requires to construct a deterministic automaton for $\mathcal{X}^{n,m}$ and then solve complex parity games. In particular, when combining determinization with the treatment of $\forall^{\approx}$ quantification, even relatively simple specifications may lead to very complex deterministic automata and (as a result) games that are complicated to solve.

Since the publication of the original Rosner reduction, several alternative approaches to asynchronous synthesis have been suggested. Vardi suggests an automata theoretic solution that shows how to embed the scheduling variable directly in the tree automaton [25]. Schewe and Finkbeiner extend these ideas to the case of branching time specifications [23]. Both approaches require the usage of determinization and the solution of general parity games. Unlike the generalized Rosner reduction they obfuscate the relation between the asynchronous and synchronous synthesis problems. In particular, the simple cases identified for asynchronous synthesis in the following sections rely on this relation between the two types of synthesis. All three approaches do not offer a practical solution to asynchronous synthesis as they have proven impossible to implement.

4 A More General Asynchronous Interaction Model

The reader may object to the model of asynchronous interaction as over simplified. Here, we justify this model by showing that it is practically equivalent (from a synthesis point of view) to a model that is more akin to software thread implementation. Specifically, we introduce a model in which the environment chooses the times the system can read or write and the system chooses whether to read or write and which variable to access. We formally define this model and show that the two asynchronous models are equivalent. We call our original asynchronous interaction model *round robin* and this new model *by demand*.

For this section, without loss of generality, fix a LTL specification $\varphi(X; Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$.

4.1 A General (Unrestricted) Model

A *by-demand program* P_b from X to Y is a function $P_b : D^* \mapsto \{1, \ldots, n\} \cup (D \times \{n+1, \ldots, n+m\})$. That is, for a given history of values read\written by the program (and the program should know which variables it read\wrote) the program decides on the next variable to read\write. In case that the decision is to write in the next $I\backslash O$ point, the program also chooses the value to write. We also assume that for $0 \leq i < m$ and for every $d_1, \ldots, d_{m-1} \in D$, we have

$P_b(d_1, \ldots, d_i) = (d, (n + i + 1))$ for some $d \in D$. That is, the program starts by writing all the output variables according to their order $y_1, y_2, \ldots, y_m$.

We define when an interaction matches a by-demand program. Recall that an interaction over X and Y is $\sigma = (a_0, b_0), (a_1, b_1), \ldots \in (D^X \times D^Y)$. An $I\backslash O$-*sequence* is $C = c_0, c_1, \ldots$ where $0 = c_0 < c_1 < c_2, \ldots$. It identifies the points in which the program reads or writes. For a sequence $d_1, \ldots, d_k \in D^*$, we denote by $t(P_b(d_1, \ldots, d_k))$ the value j such that either $P_b(d_1, \ldots, d_k) \in \{1, \ldots, n\}$ and $P_b(d_1, \ldots, d_k) = j$ or $P_b(d_1, \ldots, d_k) \in D \times \{n+1, \ldots, n+m\}$ and $P_b(d_1, \ldots, d_k) = (d, j)$. That is, $t(P_b(d_1, \ldots, d_k))$ tells us which variable the program P_b is going to access in the next $I\backslash O$-point. Given an interaction σ, an $I\backslash O$ sequence C, and an index $i \geq 0$, we define the *view* of P_b, denoted $v(P_b, \sigma, C, i)$, as follows.

$$v(P_b, \sigma, C, i) = \begin{cases} b_0[1], \ldots, b_0[m] & \text{If } i = 0 \\ v(P_b, \sigma, C, i-1), a_{c_i}[t(P_b(v(P_b, \sigma, C, i-1)))] & \\ \qquad \text{If } i > 0 \text{ and } t(P_b(v(P_b, \sigma, C, i-1))) \leq n & \\ v(P_b, \sigma, C, i-1), b_{c_i}[t(P_b(v(P_b, \sigma, C, i-1)))] & \\ \qquad \text{If } i > 0 \text{ and } t(P_b(v(P_b, \sigma, c, i-1))) > n & \end{cases}$$

That is, the view of the program is the part of the interaction that is observable by the program. The view starts with the values of all outputs at time zero. Then, the view at c_i extends the view at c_{i-1} by adding the value of the variable that the program decides to read\write based on its view at point c_{i-1}.

The interaction σ is a *by-demand asynchronous interaction* of P_b for $I\backslash O$ sequence C if the following three requirements hold:

(a) for every $1 \leq j \leq m$ we have $P_b(b_0[1], \ldots, b_0[j-1]) = (b_0[j], (n + j))$,
(b) for every $i > 1$ and every $k > 0$ such that $c_i \leq k < c_{i+1}$, we have
  - If $t(P_b(v(P_b, \sigma, C, i-1))) \leq n$, forall $j \in \{1, \ldots, m\}$ we have $b_k[j] = b_{k-1}[j]$.
  - If $t(P_b(v(P_b, \sigma, C, i-1))) > n$, forall $j \neq t(P_b(v(P_b, \sigma, C, i-1)))$ we have $b_k[j] = b_{k-1}[j]$ and for $j = t(P_b(v(P_b, \sigma, C, i-1)))$ we have $P_b(v(P_b, \sigma, c, i-1)) = (b_k[j], j)$.

(c) and for every $j \in \{1, \ldots, m\}$ and every $0 < k < c_1$, we have $b_k[j] = b_0[j]$.

That is, the interaction matches a by-demand program if (a) the interaction starts with the right values of all outputs (as the program starts by initializing them), (b) the outputs do not change in the interaction unless at $I\backslash O$ points where the program chooses to update a specific output (based on the program's view of the intermediate state of the interaction), and (c) the outputs do not change before the first $I\backslash O$ point.

Definition 2 (by-demand realizability). *A* LTL *specification* $\varphi(X; Y)$ *is* **by-demand asynchronously realizable** *if there exists a by-demand program* P_a *such that all by-demand asynchronous interactions of* P_a *(for all* $I\backslash O$*-sequences) satisfy* $\varphi(X; Y)$.

Theorem 4. *A* LTL *specification* $\varphi(X; Y)$ *is asynchronously realizable* **iff** *it is by-demand asynchronously realizable. Furthermore, given a program that asynchronously realizes* $\varphi(X; Y)$*, it can be converted in linear time to a program that by-demand asynchronously realizes* $\varphi(X; Y)$*, and vice versa.*

Proof (Sketch): A round-robin program is trivially a by-demand program.

Showing that if a specification is by-demand realizable then it is also round-robin realizable is more complicated. Given a by-demand program, a round-robin program can simulate it by waiting until it has access to the variable required by the by-demand program. This means that the round-robin program may idle when it has the opportunity to write outputs and ignore inputs that it has the option to read. However, the resulting interactions are still interactions of the by-demand program and as such satisfy the specification. □

Although the concept, as explained above, is simple, the proof is quite involved as the structure of round-robin and by-demand programs is quite different.

Proof: We shall prove both directions:

⇐ Let $\varphi(X;Y)$ be by-demand asynchronously realized by the by-demand asynchronous program $P_b : D^* \mapsto \{1,\ldots,n\} \cup (D \times \{n+1,\ldots,n+m\})$. We define the asynchronous program $P_a : (D^X)^* \mapsto D^Y$ as shown below.
For all $k \geq 0$, $a_1,\ldots,a_k \in (D^X)^k$, we define $P_a(a_1,\ldots,a_k)$ inductively, as follows. We also define inductively $v_k \in D^+$, which holds the k-th view of P_b that is used to define P_a. Set $P_a(\epsilon) = b_0$, where for all $0 < i \leq m$ $P_b(b_0[1],\ldots,b_0[i-1]) = (b_0[i],(n+i))$ (this uniquely defines $b_0 \in D^Y$). Also, let $v_0 = b_0[1],\ldots,b_0[m]$. For all $k > 0$, let $t_k = t(P_b(v_{k-1}))$. If $t_k \leq n$, define $P_a(a_1,\ldots,a_k) = P_a(a_1,\ldots,a_{k-1})$ and let $v_k = v_{k-1}, a_k[t_k]$. If, on the other hand, $t_k > n$, then let $P_b(v_{k-1}) = (d_k,t_k)$ for some $d_k \in D$ and for all $1 \leq i \leq m$, if $i = t_k$ then define $P_a(a_1,\ldots,a_k)[i] = d_k$ and otherwise $(i \neq t_k)$ define $P_a(a_1,\ldots,a_k)[i] = P_a(a_1,\ldots,a_{k-1})[i]$. Also, if $t_k > n$ then let $v_k = v_{k-1}, d_k$.
We now show that all asynchronous interactions of P_a, for all schedules, satisfy $\varphi(X;Y)$, implying that $\varphi(X;Y)$ is asynchronously realized by P_a. Let (R,W) be a schedule, and let $\sigma = (a_0,b_0),(a_1,b_1),\ldots$ be an asynchronous interaction of P_a for this schedule. Denote $R = r_1^1,\ldots,r_1^n,r_2^1,\ldots,r_2^n,\ldots$ and $W = w_1^1,\ldots,w_1^m,w_2^1,\ldots,w_2^m,\ldots$.
We note that σ is also a by-demand asynchronous interaction of P_b, where the $I\backslash O$ points are restricted to either one reading point or one writing point from $r_i^1,\ldots,r_i^n,w_i^1,\ldots,w_i^m$ for every i. More formally, let $C = c_0,c_1,\ldots$ be the $I\backslash O$ sequence that produces σ as a by-demand asynchronous interaction of P_b. We set $c_0 = 0$ and for all $i > 0$ let $t(P_b(v(P_b,\sigma,C,i-1))) = j$, and if $j \leq n$ then $c_i = r_i^j$ and if $j > n$ then $c_i = w_i^j$. Note that $t(\cdot)$ and $v(\cdot)$ above are well defined as $v(P_b,\sigma,C,i-1)$ requires only the elements of C up to the $i-1$-th element.
As σ satisfies $\varphi(X;Y)$ (by correctness of P_b), we are done.

⇒ Let $\varphi(X;Y)$ be asynchronously realized by the asynchronous program $P_a : (D^X)^* \mapsto D^Y$. We define the by-demand asynchronous program $P_b : D^* \mapsto \{1,\ldots,n\} \cup (D \times \{n+1,\ldots,n+m\})$ as shown below. Intuitively, P_b reads and writes the appropriate variables in a cyclical order, mimicking the behavior of P_a.

For a sequence $\tau = (a_1, b_1), \ldots, (a_k, b_k) \in (D^X \times D^Y)^*$ we define the *unwinding* $q(\tau)$ as the sequence of individual values for individual variables that appear in the sequence τ. We define concurrently P_a and the function $q : (D^X \cup D^Y)^* \mapsto D^+$. Let $P_a(\epsilon) = b_0$. Then, for all $0 \leq i < m$ let $d_i = b_0[i]$ and define $P_b(d_1, \ldots, d_i) = (d_{i+1}, (n + i + 1))$. Also define $P_b(d_1, \ldots, d_m) = 1$. Define $q(\epsilon) = d_1, \ldots, d_m$. For all $\tau \in (D^X \cup D^Y)^*$, $a \in D^X$ and $0 < i \leq n$ let $a[i] = d_i$ and define $q(\tau, a) = q(\tau), d_1, \ldots, d_n$. In addition, for all $\tau \in (D^X \cup D^Y)^*$, $a \in D^X$, $b \in D^Y$ and $0 < i \leq m$ let $b[i] = d_i$ and define $q(\tau, a, b) = q(\tau, a), d_1, \ldots, d_m$. In general, consider $k > 0$ and let $P_a(a_1, \ldots, a_k) = b_k$, where for all $0 < i \leq n$ we have $a_k[i] = d_i$ and for all $n < i \leq n+m$ we have $b_k[i-n] = d_i$. Then, for every $0 < i < n$ we set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_i) = i + 1$, for every $n \leq i < n + m$ we set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_i) = (d_{i+1}, (i + 1))$, and we finally set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_{n+m}) = 1$.

We now show that all by-demand asynchronous interactions of P_b, for all $I\backslash O$-sequences, satisfy $\varphi(X; Y)$, implying that $\varphi(X; Y)$ is by-demand asynchronously realized by P_b. Let C be an $I\backslash O$ sequences, and let $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be a by-demand asynchronous interaction of P_b for this $I\backslash O$ sequences. Denote $C = c_0, c_1, c_2, \ldots$, where $c_0 = 0$.

We define a schedule (R, W), where $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$, as follows. For all $i > 0$, consider the unique j and $0 < k \leq n + m$ such that $c_i = c_{j\cdot(n+m)+k}$. If $k \leq n$ then define r_{j+1}^k, and otherwise (if $n < k$) define w_{j+1}^{k-n}. This defines (R, W) completely. We note that σ is also an asynchronous interaction of P_a for (R, W).

As σ satisfies $\varphi(X; Y)$ (by correctness of P_a), we are done.

□

4.2 A Modified Generalized Rosner Reduction

Although by-demand asynchronous synthesis can be reduced to round-robin asynchronous synthesis, we believe that techniques for handling by-demand synthesis directly will be more efficient in practice. Hence, inspired by the Rosner reduction from Subsection 2.4, we propose a family of translations (one translation for each pair (n, m)) called the *by-demand generalized Rosner reduction*. We translate a specification $\varphi(X; Y)$ into a QPTL specification $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ that has an additional Boolean input variable c and an additional output variable h that ranges over $\{1, \ldots, (n + m)\}$. The new variable c is called the *Boolean $I\backslash O$ variable*, and h is called the *$I\backslash O$-selector variable*.

The role of c is similar to the role of r in the previous reduction. A change in the value of c indicates an $I\backslash O$-point. The value of h indicates the choice of which variable to read\write. Values $1, \ldots, n$ indicate reading and values $(n + 1), \ldots, (n+m)$ indicate writing. We set a new value for h right after a read\write. Thus, the system immediately commits to the next variable it is going to access.

As h is treated like all other outputs, the system cannot change its value when c does not change. This corresponds to no new information being gained by the system as long as c does not change.

In this section, we reuse the notations $\mathit{unchanged}(x)$ to indicate a state where variable x did not change its value, and *first* to indicate a state that is the first one in the computation.

The kernel formula $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\}$ is defined as follows.

$$\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\}) = \gamma(c) \rightarrow \delta^{n,m}(X \cup \{c\}; Y \cup \{h\}),$$

where

$$\gamma(c) = \bar{c} \wedge \Box \Diamond c \wedge \Box \Diamond \bar{c}$$

and $\delta^{n,m}(X \cup \{c\}; Y \cup \{h\})$ is given by

$$\begin{pmatrix} [(c \leftrightarrow \ominus c) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(h) & \wedge \\ \varphi(X;Y) & \wedge \\ \bigwedge_{i=1}^{m} \left\{ \left[\left[(c \leftrightarrow \ominus c) \vee \ominus [h \neq (i+n)] \right] \wedge \neg \mathit{first} \right] \Rightarrow \mathit{unchanged}(y_i) \right\} & \wedge \\ (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} \left[\left[\neg (c \leftrightarrow \ominus c) \wedge \ominus (h = i) \right] \Rightarrow (x_i = \tilde{x_i}) \right] \rightarrow \varphi(\tilde{X}; Y) & \end{pmatrix}.$$

The initial value of the $I\backslash O$-selector variable h is selected nondeterministically.

Similar to the role of the scheduling variable in $\mathcal{X}^{n,m}$, the variables c and h in $\mathcal{Y}^{n,m}$ make explicit the decisions of the environment when to have an $I\backslash O$ point for the system (c), and which variable to read\write (h).

Theorem 5. *Let $\varphi(X;Y)$ be a* LTL *specification where $|X| = n$ and $|Y| = m$, and let its kernel formulae be $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ and $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, derived by the by-demand generalized Rosner reduction, and by the generalized Rosner reduction, respectively.*

The specification $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ is synchronously realizable **iff** *the specification $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable. Furthermore, given a program $P_{\mathcal{Y}}$ that synchronously realizes $\mathcal{Y}^{n,m}$, it can be converted to a program $P_{\mathcal{X}}$ that synchronously realizes $\mathcal{X}^{n,m}$ in time linear in the number of transitions of the LTS that induces $P_{\mathcal{X}}$, and vice versa.*

Proof: We shall prove both directions:

$\Leftarrow$ Having that $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable, means that there exists a program $P_{\mathcal{X}}$ that synchronously realizes it. We describe the construction of another program, $P_{\mathcal{Y}}$, that synchronously realizes $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, effectively proving that it is synchronously realizable.
To do this, we need to provide a function that corresponds to $P_{\mathcal{Y}}$, and that generates assignments to $Y \cup \{h\}$, given finite histories of assignments to $X \cup \{c\}$. We then need to prove that all synchronous interaction of this program which we construct, satisfy $\mathcal{Y}^{n,m}$.

Let $\sigma_{X,c} = \sigma_0, \sigma_1, \ldots, \sigma_k$ be a finite history of assignments to $X \cup \{c\}$ for $k \geq 0$, such that for $0 \leq i \leq k$, $\sigma_i = (X_i, c_i)$ where X_i is an assignment to X and c_i is an assignment to c. We construct the sequence of assignments to r, $\eta_r = r_0, r_1, \ldots, r_k$, in the following way: $r_0 = n + m$. For $k \geq i > 0$, if $c_i \leftrightarrow \ominus c_{i-1}$ then $r_i = r_{i-1}$, otherwise $r_i = r_{i-1} \oplus_{n+m}$.
Let an assignment to $Y \cup \{h\}$ be a pair (Y_i, h_i) where Y_i is an assignment to Y and h_i is an assignment to h. Similarly, an assignment to $X \cup \{r\}$ is a pair (X_i, r_i) where X_i is an assignment to X and r_i is an assignment to r.
We define

$$P_{\mathcal{Y}}(\sigma_{X,c}) = \left(\underbrace{P_{\mathcal{X}}\Big((X_0, r_0), (X_1, r_1), \ldots, (X_k, r_k)\Big)}_{Y_k}, \underbrace{r_k \oplus_{n+m}}_{h_k} \right)$$

where Y_k is an assignment to Y and h_k is an assignment to h.
It is not difficult to see why any computations that would be based on a synchronous interaction with $P_{\mathcal{Y}}$ would satisfy $\mathcal{Y}^{n,m}$. If in a particular computation $\mu_{X,c}$ the $I \backslash O$ variable c does not change its value infinitely often, then $\mathcal{Y}^{n,m}$ is trivially satisfied. Otherwise, by the way we construct μ_r (a computation for $\{r\}$), $\mu_r, 0 \models \alpha^{n,m}(r)$, and we know that $\mu_{X,r,Y}$ (using the Y values that we output from $P_{\mathcal{X}}$) satisfies all three conjuncts of $\beta^{n,m}(X \cup \{r\}; Y)$. The first one, $\varphi(X; Y)$, appears also in $\mathcal{Y}^{n,m}$. The second one turns out to be essentially identical to the third conjunct in $\delta^{n,m}$, since h is identical to r at all $I \backslash O$ points, and since c and r change their values always together. The last conjuncts in both $\beta^{n,m}$ and $\delta^{n,m}$ are essentially identical for the same reasons. The remaining conjunct in $\delta^{n,m}$, the first one, is also satisfied by h starting with the value $r_0 \oplus_{n+m} = (n + m) \oplus_{n+m} = 1$, and by $[(c \leftrightarrow \ominus c) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(h)$ holding due to the fact that we change r **iff** c changes, and we change h **iff** r changes.

$\Rightarrow$ In this direction we know that there exists a program $P_{\mathcal{Y}}$ that synchronously realizes $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, and we construct a program $P_{\mathcal{X}}$ that synchronously realizes $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. We use similar notations of assignments and histories as in the other direction.
Let $\sigma^k_{X,r} = \sigma_0, \sigma_1, \ldots, \sigma_k$ be a prefix for $k \geq 0$ of a computation over the variables in $X \cup \{r\}$ (we freely use similar notations for other sets of variables). If $r_0 \neq (n + m)$, or if there exists some index $i > 0$ such that $(r_i \neq r_{i-1}) \wedge [r_i \neq (r_{i-1} \oplus_{n+m})]$ holds, then for all $j \geq i$ (for all $j \geq 0$ if $r_0 \neq (n+m)$) we define $Y_j = Y_{rand}$ for some arbitrary $Y_{rand} \in D^Y$. From this point onwards in the construction we assume that this is not the case, and that r updates as indicated by $\alpha^{n,m}(r)$. We construct the prefixes $\sigma^k_{c,Y,h}$, for all k, inductively (using the initial value of h, h_0, which is deterministically selected by $P_{\mathcal{Y}}$):

- Let i_1 be the minimal index such that r_{i_1} changes its value to h_0 (so that $r_{i_1} = h_0$). We know that there exists $i_1 > 0$ since we assume that r changes cyclically infinitely often. Let $\sigma^{i_1}_c = c_0, \ldots, c_{i_1}$, where $\overline{c_0}$ holds, and for $0 < j < i_1$ $\overline{c_j}$ holds. c_{i_1} holds as well ($c_{i_1} = \mathrm{T}$). For $0 \leq j \leq i_1$ $P_{\mathcal{Y}}((X_0, c_0), \ldots, (X_j, c_j)) = (Y_j, h_j)$.

- Let i_t be the minimal index that is greater than i_{t-1} such that r_{i_t} changes its value to $h_{i_{t-1}}$ (so that $r_{i_t} = h_{i_{t-1}}$). We know that there exists such i_t. Let $\sigma_c^{i_t} = \sigma_c^{i_{t-1}}, c_{i_{t-1}+1}, \ldots, c_{i_t}$, where for $i_{t-1}+1 \leq j < i_t$ $c_j \leftrightarrow c_{i_{t-1}}$ holds. $\neg(c_{i_t} \leftrightarrow c_{i_{t-1}})$ holds as well. For $i_{t-1} < j \leq i_t$ $P_{\mathcal{Y}}((X_0, c_0), \ldots, (X_j, c_j)) = (Y_j, h_j)$.

Using the construction and definitions described above, we define

$$P_{\mathcal{X}}(\sigma_{X,r}^k) = Y_k$$

where Y_k is an assignment to Y.

It is not difficult to see why any computations that would be based on a synchronous interaction with $P_{\mathcal{X}}$ would satisfy $\mathcal{X}^{n,m}$. If in a particular computation $\mu_{X,r}\ \mu_r, 0 \not\models \alpha^{n,m}(r)$ then $\mathcal{X}^{n,m}$ is trivially satisfied with Y_{rand}. Otherwise, we can definitely construct σ_c^k for all k (since r changes cyclically infinitely often, and therefore admits all of its domain values infinitely often). Since in each iteration of σ_c^k construction we have exactly one change of c value, then $\mu_c, 0 \models \gamma(c)$. Since we constructed $\mu_{Y,h}$ using $P_{\mathcal{Y}}$, we get that $\mu_{X,c,Y,h}$ satisfies all four conjuncts of $\delta^{n,m}(X \cup \{c\}; Y \cup \{h\})$. The second one, $\varphi(X;Y)$, appears also in $\mathcal{X}^{n,m}$. Noticing that the set of states in $\mu_{X,c,r,Y,h}$ that satisfy $(c \leftrightarrow \ominus c) \vee \ominus[h \neq (i+n)]$ is a super-set of the states that satisfy $\neg write_n(i)$ (for all $i \in \{1, \ldots, m\}$), we get that the satisfaction of the third conjunct in $\delta^{n,m}$ guarantees the satisfaction of the second conjunct in $\beta^{n,m}$. Finally, since the set of states in $\mu_{X,c,r,Y,h}$ that satisfy $\neg(c \leftrightarrow \ominus c) \wedge \ominus(h = i)$ is a sub-set of the states that satisfy $read(i)$ (again, for all $i \in \{1, \ldots, m\}$), we get that the satisfaction of the last conjunct in $\delta^{n,m}$ guarantees the satisfaction of the last conjunct in $\beta^{n,m}$ (since there are fewer reading points in $\mathcal{Y}^{n,m}$, then $\varphi(\tilde{X};Y)$ must hold for a large set of computations $\tilde{X}$, including all of those that are 'allowed' by the clause $\varphi(\tilde{X};Y)$ in $\mathcal{X}^{n,m}$).

□

Theorem 6. *Given a specification $\varphi(X;Y)$ ($|X| = n$, and $|Y| = m$), the following conditions are equivalent:*

1. *$\varphi(X;Y)$ is by-demand asynchronously realizable.*
2. *$\varphi(X;Y)$ is asynchronously realizable.*
3. *The kernel formula $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, which is derived from $\varphi(X;Y)$ using the by-demand generalized Rosner reduction, is synchronously realizable.*
4. *The kernel formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, which is derived from $\varphi(X;Y)$ using the generalized Rosner reduction, is synchronously realizable.*

Furthermore, given a program P that realizes one of these specifications, it can be converted to a program that realizes any of the other in time linear in the number of transitions of the LTS/ILTS that induces P.

Proof: This is a direct result of Theorem 3, Theorem 4, and Theorem 5. □

Theorem 6 finally confirms that both $\mathcal{Y}^{n,m}$ and $\mathcal{X}^{n,m}$ may be freely used to test for any type of asynchronous realizability of $\varphi(X;Y)$, as well as for synthesis. From this point onward we consider only round-robin asynchronous realizability and the reduction from $\varphi(X;Y)$ to $\mathcal{X}^{n,m}$.

5 Proving Unrealizability of a Specification

In this section we show how an over-approximation of $\mathcal{X}^{n,m}$ can effectively prove that a given specification is asynchronously unrealizable.

5.1 Over-Approximating the Kernel Formula

Fix an LTL specification $\varphi(X;Y) = \mathit{Imp}(\varphi_e, \varphi_s)$. Let $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and let r be a scheduling variable ranging over $\{1, \ldots, (n+m)\}$. Let $\tilde{X} = \{\tilde{x} | x \in X\}$. We assume that $r \notin X \cup Y$ and that $\tilde{X} \cap (X \cup Y) = \emptyset$.

In this section, we reuse the notations $\mathit{write}_n(i)$ to indicate a state that is a writing point for the i'th output, $\mathit{read}(i)$ to indicate a state that is a reading point for the i'th input, $\mathit{unchanged}(x)$ to indicate a state where variable x did not change its value, and *first* to indicate a state that is the first one in the computation.

As explained in Section 3, the generalized Rosner reduction, although offering a complete solution to the asynchronous synthesis problem, is often prohibitively costly to use due to the universal quantification in the clause $\beta_3^{n,m}$. If we wish to use 'effective' algorithms for synthesis based on this reduction, we must find a way to avoid the size increase caused by $\beta_3^{n,m}$.

Recall the generalized Rosner reduction formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. As noted in Subsection 2.4, the first conjunct in $\beta^{n,m}$ – $\varphi(X;Y)$ – is a redundant one that was left in the Rosner reduction and in its generalizations only for clarity purposes (this conjunct follows from $\beta_3^{n,m}$). Therefore, in this section we allow ourselves to remove $\varphi(X;Y)$ from $\beta^{n,m}$, leaving us with

$$\tilde{\mathcal{X}}^{n,m}(X \cup \{r\}; Y) = \alpha^{n,m}(r) \rightarrow \tilde{\beta}^{n,m}(X \cup \{r\}; Y)$$

where $\tilde{\beta}^{n,m}(X \cup \{r\}; Y)$ is given by

$$\left(\begin{array}{l} \bigwedge_{i=1}^{m} \Big[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \mathrel{\tilde{\Rightarrow}} \mathit{unchanged}(y_i) \Big] \quad \wedge \\ (\forall^{\approx} \tilde{X}). \Big[\bigwedge_{i=1}^{n} [\mathit{read}(i) \mathrel{\tilde{\Rightarrow}} (x_i = \tilde{x_i})] \Big] \rightarrow \varphi(\tilde{X}; Y) \end{array} \right).$$

We know that $\tilde{\mathcal{X}}^{n,m} \leftrightarrow \mathcal{X}^{n,m}$, and we may use them interchangeably. We still use $\beta_3^{n,m}$ as a name for the last conjunct of $\beta^{n,m}$ and of $\tilde{\beta}^{n,m}$

With this in mind, we define an *over-approximating formula* for $\mathcal{X}^{n,m}$:

$$\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y) = \alpha^{n,m}(r) \rightarrow \beta_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$$

where $\beta^{n,m}_{\downarrow}(X \cup \tilde{X} \cup \{r\}; Y)$ is given by

$$\left(\begin{array}{l} \bigwedge_{i=1}^{m} \Big[[\neg write_n(i) \wedge \neg first] \twoheadrightarrow unchanged(y_i) \Big] \wedge \\ \Big[\bigwedge_{i=1}^{n} [read(i) \twoheadrightarrow (x_i = \tilde{x_i})] \Big] \rightarrow \varphi(\tilde{X}; Y) \end{array} \right).$$

Note that $\mathcal{X}^{n,m}_{\downarrow}$ is almost identical to $\tilde{\mathcal{X}}^{n,m}$, except that the second clause in $\beta^{n,m}_{\downarrow}$ has no quantification (over $\tilde{X}$), eliminating the source of trouble in $\beta^{n,m}_{3}$. In effect, this amounts to adding $\tilde{X}$ to the set of input variables. In fact, if $\varphi(X; Y)$ has a $GR(1)$ winning condition, then it could be easily shown by propositional arguments[2] that $\mathcal{X}^{n,m}_{\downarrow}$ has a $GR(1)$ winning condition as well. The idea is now to use $\mathcal{X}^{n,m}_{\downarrow}$ to deduce unrealizability of $\mathcal{X}^{n,m}$. As $\mathcal{X}^{n,m}_{\downarrow}$ does not include quantification, we could use the effective algorithm of [16] on $\mathcal{X}^{n,m}_{\downarrow}$.

The main observation relating $\mathcal{X}^{n,m}$ to $\mathcal{X}^{n,m}_{\downarrow}$ is the following theorem:

Theorem 7. *For a specification $\varphi(X; Y)$ where $|X| = n$ and $|Y| = m$, and for a scheduling variable r ranging over $\{1, \ldots, (n+m)\}$, the following holds: If $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable, then $\mathcal{X}^{n,m}_{\downarrow}(X \cup \tilde{X} \cup \{r\}; Y)$ is synchronously realizable.*

Proof: Let P_s be a program that synchronously realizes $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. We shall prove that P_s also synchronously realizes $\mathcal{X}^{n,m}_{\downarrow}(X \cup \tilde{X} \cup \{r\}; Y)$.

Let σ be a computation that is induced by $\mathcal{X}^{n,m}$, so that $\sigma, 0 \models \mathcal{X}^{n,m}$. We would like to show that $\sigma, 0 \models \mathcal{X}^{n,m}_{\downarrow}$.

If $\sigma, 0 \not\models \alpha^{n,m}$, then, trivially, $\sigma, 0 \models \mathcal{X}^{n,m}_{\downarrow}$. Otherwise, we know that σ satisfies also the first conjunct in $\tilde{\beta}^{n,m}$ which appears in $\beta^{n,m}_{\downarrow}$. To prove that $\sigma, 0 \models \mathcal{X}^{n,m}_{\downarrow}$, we are left with proving that σ satisfies the last (second) conjunct in $\beta^{n,m}_{\downarrow}$.

We also know, however, that σ satisfies the last conjunct in $\beta^{n,m}$: $\sigma, 0 \models (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} [read(i) \twoheadrightarrow (x_i = \tilde{x_i})] \rightarrow \varphi(\tilde{X}; Y)$. Since this means that the implication that appears in this conjunct would be satisfied by any $\tilde{X}$-variant σ'' of any stuttering variant σ' of σ, we only weaken this statement by writing that $\sigma, 0 \models \bigwedge_{i=1}^{n} [read(i) \twoheadrightarrow (x_i = \tilde{x_i})] \rightarrow \varphi(\tilde{X}; Y)$. Since this is a weakening transition (claiming satisfiability by σ only), it is correct. This is, however, exactly the last conjunct in $\beta^{n,m}_{\downarrow}$, and the proof is complete. □

An important result of Theorem 7 is the following:

[2] Roughly speaking, all elements of $\mathcal{X}^{n,m}_{\downarrow}$ could be 'absorbed' into $\varphi(\tilde{X}; Y)$ without increasing the formula's complexity in terms of the temporal hierarchy.

Theorem 8. *For a specification $\varphi(X;Y)$ where $|X| = n$ and $|Y| = m$, and for a scheduling variable r ranging over $\{1,\ldots,(n+m)\}$, the following holds: If $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\};Y)$ is synchronously unrealizable, then $\varphi(X;Y)$ is asynchronously unrealizable.*

Proof: This is a direct result of Theorem 1, Theorem 3 and Theorem 7. □

Theorem 8 provides us with the framework for an effective way to test whether specifications with $GR(1)$ winning conditions are asynchronously unrealizable, as desired. This is what justifies referring to $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\};Y)$ as an **over-approximation** (equivalently, weakening) of $\mathcal{X}^{n,m}(X \cup \{r\};Y)$. Indeed, the effective algorithm of [16] could be used with $\mathcal{X}_{\downarrow}^{n,m}$ to test whether its underlying specification $\varphi(X;Y)$ is asynchronously unrealizable. The time complexity of using this algorithm (for specifications with $GR(1)$ winning conditions) is $O(N^3 \cdot m \cdot n)$, where N is the state space of the specification, and m and n are the number of liveness conjuncts of the environment and system's specifications, respectively.

Caveat: We refer the reader to [11], which exposes a flaw in [16]. In general, the algorithm in [16] may declare a specification synchronously unrealizable, while they are in fact realizable. The work in [11] shows how to bypass this problem.

Note that while the methods proposed in this section are sound, they are not complete, in the sense that a specification $\varphi(X;Y)$ may be asynchronously unrealizable but the derived synchronous approximation $\mathcal{X}_{\downarrow}^{n,m}$ may be synchronously realizable.

5.2 Applying the Unrealizability Test

In this subsection we illustrate the application of the effective unrealizability test based on Theorem 8.

We start with the 'copy' specification $\varphi_1(x;y) : \square(x \leftrightarrow y)$ which we considered in the introduction (both x and y are Booleans, and for clarity, we omit the initial condition - $\overline{x} \wedge \overline{y}$). We claimed that this specification is asynchronously unrealizable but stated this fact with no proof. Now, we have an adequate tool for proving that this specification is indeed asynchronously unrealizable. Deriving the kernel formula $\mathcal{X}_{\downarrow}^{1,1}(x,\tilde{x},r;y)$ for $\varphi_1(x;y)$, we obtain the specification $\mathcal{X}_{\downarrow}\langle\varphi_1\rangle = \alpha^{1,1}(r) \rightarrow \beta_{\downarrow}\langle\varphi_1\rangle$, where $\beta_{\downarrow}\langle\varphi_1\rangle(x,\tilde{x},r;y)$ is given by

$$\begin{pmatrix}(((r \neq 2) \vee \ominus(r \neq 1)) \wedge \neg \mathit{first}) \mathrel{\Rightarrow\!\!\!\!\succ} \mathit{unchanged}(y) \wedge \\ (((r = 1) \wedge \ominus(r \neq 1)) \mathrel{\Rightarrow\!\!\!\!\succ} (x \leftrightarrow \tilde{x})) \rightarrow \square(\tilde{x} \leftrightarrow y)\end{pmatrix}.$$

We proceed to show that there can be no synchronous program that satisfies (by controlling y) $\mathcal{X}_{\downarrow}\langle\varphi_1\rangle(x,\tilde{x},r;y)$ for all choices of x, $\tilde{x}$, and r. Assume the opposite, and consider a computation $\sigma = a_0, a_1, \ldots$, such that $\overline{x}$ holds at all states, and $\tilde{x}$ and $r = 2$ hold at a state a_j **iff** j is odd. It follows that all even

indexed states are reading points, and $x = \tilde{x}$ at all of these states. Consequently, and since $\sigma, 0 \models \alpha^{1,1}(r)$, we should have $\sigma, 0 \models \square(y \leftrightarrow x)$ and $\sigma, 0 \models \square(\tilde{x} \leftrightarrow y)$. However, this implies that $x \leftrightarrow \tilde{x}$ should hold at all states, which is false because x and $\tilde{x}$ differ at all odd-indexed states.

We conclude that the specification $\varphi_1(x; y) : \square(x \leftrightarrow y)$ is asynchronously unrealizable. Indeed, when checking synchronous realizability of the kernel formula $\mathcal{X}^{1,1}_{\downarrow}(x, \tilde{x}, r; y)$ for $\varphi_1(x; y)$ using the algorithm of [16], we get that it is unrealizable.

Assume that we are not ready to give up and would like to develop an asynchronous system that captures some of the essential behavior of a copying module. An informal description of such a behavior can include the following requirements:

1. Whenever x rises to 1, then sometimes later y should rise to 1.
2. Whenever x drops to 0, then sometimes later y should drop to 0.
3. Variable y should not rise to 1, unless sometimes before x was 1.
4. Variable y should not drop to 0, unless sometimes before x was 0.

A temporal formula that captures these four requirements may be given by the following specification:

$$\varphi_2(x; y) : \left(\begin{array}{l} (x \Rightarrow \diamondsuit\, y) \wedge (\overline{x} \Rightarrow \diamondsuit\, \overline{y}) \wedge \\ (y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x) \wedge \bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x}) \end{array} \right)$$

As before, both x and y are Booleans, and for clarity, we omit the initial condition - $\overline{x} \wedge \overline{y}$. The past formula $y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x$ states that if currently y holds then this state was preceded by an interval in which y continuously held, preceded by an interval in which $\overline{y}$ continuously held, preceded by a state at which x held. The formula $\bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x})$ states that, starting at the second state, if currently $\overline{y}$ holds, then this state is preceded by an interval in which $\overline{y}$ continuously held, and which either extends to the beginning of the computation or is preceded by an interval in which y continuously held and which is preceded by a state at which $\overline{x}$ held.

We will now apply the unrealizability test to check whether $\varphi_2(x; y)$ is also asynchronously unrealizable. In order to conclude that this is the case, we have to find a computation $\sigma = a_0, a_1, \ldots$ in which x and $\tilde{x}$ agree infinitely often (on reading points), and where, regardless of y values, one of the following must hold: $\sigma, 0 \not\models \varphi_2(x; y)$ or $\sigma, 0 \not\models \varphi_2(\tilde{x}; y)$. Assume that $\overline{x}$ holds at all states, and let $\tilde{x}$ hold at state a_j **iff** j is odd. Thus, we can take all even-indexed states to be the reading points, and $x = \tilde{x}$ at all of these states. $\sigma, 0 \models \varphi_2(x; y)$ implies that $\overline{y}$ holds at all states. This is since any occurrence of y at some state implies, by $y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x$, that x holds at some earlier state, which never is the case. However, in this case, the fact that $\tilde{x}$ holds at state a_1 entails that $\sigma, 0 \not\models \varphi_2(\tilde{x}; y)$ because it violates the requirement $\tilde{x} \Rightarrow \diamondsuit\, y$, which is part of $\varphi_2(\tilde{x}; y)$. We conclude that $\varphi_2(x; y)$ is also asynchronously unrealizable. Again, when checking synchronous realizability of the kernel formula $\mathcal{X}^{1,1}_{\downarrow}(x, \tilde{x}, r; y)$ for $\varphi_2(x; y)$ using the algorithm of [16], we get that it is unrealizable.

How can we weaken $\varphi_2(x;y)$ into a specification that stands a better chance of being asynchronously realizable? Obviously, the weakness of the specification $\varphi_2(x;y)$ is that it allows the environment to modify x too quickly without waiting for an evidence that the system has noticed the most recent change. We can correct this drawback by allowing the environment to modify x only at points in which $x \leftrightarrow y$ (that is, after the system had enough time to respond to a change of x). For example, we can suggest the following 'response' specification:

$$\varphi_3(x;y) : [\neg(x \leftrightarrow y) \twoheadrightarrow (x \leftrightarrow \bigcirc x)] \rightarrow \begin{pmatrix} x \twoheadrightarrow \diamondsuit y & \wedge \\ \overline{x} \twoheadrightarrow \diamondsuit \overline{y} & \wedge \\ y \twoheadrightarrow y \,\mathcal{S}\, \overline{y} \,\mathcal{S}\, x & \wedge \\ \bigcirc(\overline{y} \twoheadrightarrow \overline{y} \,\mathcal{B}\, y \,\mathcal{S}\, \overline{x}) & \end{pmatrix}$$

As before, both x and y are Booleans, and for clarity, we omit the initial condition $\overline{x} \wedge \overline{y}$. Applying the unrealizability test to $\varphi_3(x;y)$, we find that its corresponding kernel formula $\mathcal{X}^{1,1}_{\downarrow}(x, \tilde{x}, r; y)$ is synchronously realizable. However, we cannot infer any conclusions from this, since Theorem 8 offers only conclusions in the unrealizable case. In the next section, we consider methods that can lead to effective realizability and apply them to the specification $\varphi_3(x;y)$.

6 Proving Realizability of a Specification, and Synthesis

As mentioned, the formula $\mathcal{X}^{n,m}$ does not lead to a practical solution for asynchronous synthesis. Here we show that in some cases a simpler synchronous realizability test can still imply the realizability of an asynchronous specification. We show that when a certain strengthening can be found and certain conditions hold with respect to the specification we can apply a simpler realizability test maintaining the structure of the specification. In particular, this simpler realizability test does not require stuttering quantification. When the original formula's winning condition is a $GR(1)$ formula, the synthesis algorithm in [16] can be applied, bypassing much of the complexity involved in synthesis[3].

We fix a specification $\varphi(X;Y) = \mathit{Imp}(\varphi_e, \varphi_s)$ with a $GR(1)$ winning condition, where $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and $\varphi_e = \langle I_{\varphi_e}, S_{\varphi_e}, L_{\varphi_e} \rangle$. Let r be a scheduling variable ranging over $\{1, \ldots, (n+m)\}$ and let $\tilde{X} = \{\tilde{x} | x \in X\}$. We define the set of *declared output variables* $\tilde{Y} = \{\tilde{y} | y \in Y\}$. We assume that $r \notin X$, $\tilde{X} \cap Y = \emptyset$, and that $\tilde{Y} \cap X = \emptyset$. We re-use the notations $\mathit{write}_n(i)$, $\mathit{read}(i)$, $\mathit{unchanged}(x)$, and *first*.

We start with a definition of a strengthening, which is a formula of the type $\psi(X, r; Y)$. Intuitively, the strengthening refers explicitly to a scheduling variable r and should imply the truth of the original specification and ignore the input except in reading points so that the stuttering quantification can be removed.

Definition 3 (asynchronous strengthening). *A specification $\psi(X, r; Y) = \mathit{Imp}(\psi_e, \psi_s)$ with a $GR(1)$ winning condition, where $\psi_e = \langle I_{\psi_e}, S_{\psi_e}, L_{\psi_e} \rangle$, is an*

[3] As before, this algorithm should be 'corrected' as described in [11]

asynchronous strengthening *of* $\varphi(X;Y)$ *if* $I_{\psi_e} = I_{\varphi_e}$, $S_{\psi_e} = S_{\varphi_e}$, *and the following implication is valid:*

$$\left(\begin{array}{ll} \alpha^{n,m}(r) & \wedge \\ I_{\psi_e} \wedge \Box\, S_{\psi_e} & \wedge \\ \psi(X, r; Y) & \wedge \\ \bigwedge_{i=1}^{n} [\mathit{read}(i) \Rightarrow (x_i = \tilde{x_i})] & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(y_i)\right] & \end{array}\right) \rightarrow \varphi(\tilde{X}; Y).$$

Checking the implication in this definition requires to check identity of propositional formulae and validity of a LTL formulae, which is supported, e.g., by JTLV [20].

The formula needs to satisfy two more conditions, which are needed to show that the simpler synchronous realizability test (introduced below) is sufficient. Stuttering robustness is very natural for asynchronous specifications as we expect the system to be completely unaware of the passage of time. Memory-lessness requires that the system knows the entire 'state' of the environment.

Definition 4 (stuttering robustness). *A* LTL *specification* $\xi(X;Y)$ *is* **stutteringly robust** *if for all computations* σ *and* σ' *such that* σ' *is a stuttering variant of* σ, $\sigma, 0 \models \xi$ **iff** $\sigma', 0 \models \xi$.

We can test stuttering robustness by converting a specification to a nondeterministic Büchi automaton [26], adding to it transitions that capture all stuttering options [19], and then checking that it does not intersect the automaton for the negation of the specification. In our case, when handling formulae with $GR(1)$ winning conditions, in many cases, all parts of the specifications are relatively simple and stuttering robustness can be easily checked.

Definition 5 (memory-lessness). *A* LTL *specification* ξ *is* **memory-less** *if for all computations* $C = c_0, c_1, \ldots$ *and* $C' = c'_0, c'_1, \ldots$ *such that* $C, 0 \models \xi$ *and* $C', 0 \models \xi$, *if for some* i *and* j *we have* $c_i = c'_j$, *then the computation* $c_0, c_1, \ldots, c_i, c'_{j+1}, c'_{j+2}, \ldots$ *also satisfies* ξ.

Specifications of the form $\varphi_e = \langle I_e, S_e, L_e \rangle$ are always memory-less. The syntactic structure of S_e forces a relation between possible current and next states that does not depend on the past. Furthermore L_e is a conjunction of properties of the form $\Box \diamondsuit p$, where p is a Boolean formula. If the specification includes past temporal operators, these are embedded into the variables of the environment (c.f. [21]), and must be accessible by the system as well.

In the general case, memory-lessness of a specification $\varphi(X;Y)$ can be checked as follows. We convert both ξ and $\neg\xi$ to nondeterministic Büchi automata N_+ and N_-. Then, we create a nondeterministic Büchi automaton A that runs two copies of N_+ and one copy of N_- simultaneously. The two copies of N_+ 'guess' two computations that satisfy $\varphi(X;Y)$ and the copy of N_- checks that the two

computations can be combined in a way that does not satisfy $\varphi(X;Y)$. Thus, the language of A would be empty **iff** $\varphi(X;Y)$ is not memory-less.

Note that if $\varphi(X;Y)$ has a memory-less environment then every asynchronous strengthening of it has a memory-less environment. This follows from the two sharing the initial and safety parts of the specification.

The following kernel formula *under-approximates* the original:

$$\mathcal{X}_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) = \alpha^{n,m}(r) \rightarrow \beta_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$$

where $\beta_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$ is given by

$$\begin{pmatrix} \mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y}) & \wedge \\ \psi(X \cup \{r\}; Y) & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \twoheadrightarrow \mathit{unchanged}(y_i) \right] & \end{pmatrix}$$

and where $\mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y})$ is given by

$$\begin{pmatrix} \bigwedge_{i=1}^{m} [\mathit{write}_n(i) \twoheadrightarrow (y_i = \tilde{y_i})] & \wedge \\ \left[\left[(r = \ominus r) \vee \bigvee_{i=1}^{m} [r = (n+i)] \right] \twoheadrightarrow \left[\bigwedge_{i=1}^{m} (\tilde{y_i} = \ominus \tilde{y_i}) \right] \right] & \end{pmatrix}.$$

The formula $\mathit{declare}^{n,m}$ ensures that the declared outputs are updated only at reading points. Indeed, for every i, $\tilde{y_i}$ is allowed to change only when r changes to a value in $\{1, \ldots, n\}$. Furthermore, the outputs themselves copy the value of the declared outputs (and only when they are allowed to change). Thus, the system 'ignores' inputs that are not at reading points in its next update of outputs.

We note that restricting to one input is similar to allowing the system to read multiple inputs simultaneously.

In the case that φ has a $GR(1)$ winning condition then so does $\mathcal{X}_{\psi}^{1,m}$. It follows that in such cases we can use the algorithm of [16] to check whether $\mathcal{X}_{\psi}$ is synchronously realizable and to extract a program that realizes it. We show how to convert a LTS realizing $\mathcal{X}_{\psi}$ to an ILTS realizing φ.

For a LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s \rangle$, state $st_{es} \in S_s$ is an *eventual successor* of state $st \in S_s$ if there exists $m \leq |S_s|$ and states $\{s_1, \ldots, s_m\} \subseteq S_s$ such that the following hold: $s_1 = st$ and $s_n = st_{es}$; For all $0 < i < m$, $(s_i; s_{i+1}) \in R_s$; For all $0 < i < m$, if $L(s_1)|_{\{r\}} = r_1$ then $L(s_i)|_{\{r\}} = r_1$, but $L(s_m)|_{\{r\}} \neq r_1$. If $L(s_m)|_{\{r\}} = 1$ we also call st_{es} an *eventual read successor*, otherwise an *eventual write successor*. Note that the way the scheduling variable r updates its values is uniform across all eventual successors of a given state.

Given a LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s \rangle$ such that $Y = \{y_1, \ldots, y_m\}$ the algorithm in Fig. 1 *extracts* from it an ILTS $T_a = \langle S_a, I_a, R_a, \{x\}, Y, L_a, i_a \rangle$. In the first part of the algorithm that follows its initialization, between lines 5 and 15, all reading states reachable from I_s are found, and used to build I_a (as part of S_a). In the second part, between lines 16 and 43, the $(m{+}1)$-th eventual successors of each reading state are added to S_a. This second part ensures that all writing states are 'skipped' so that R_a transitions include only transitions between consecutive reading states.

Input: LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s \rangle$ such that $|Y| = m$, and an initial outputs assignment Y_{init}.
Output: The elements i_a, I_a, L_a, S_a and R_a of the extracted ILTS $T_a = \langle S_a, I_a, R_a, \{x\}, Y, L_a, i_a \rangle$.

1: $i_a \leftarrow Y_{init}$
2: $I_a \leftarrow \emptyset$, $S_a \leftarrow \emptyset$, $R_a \leftarrow \emptyset$
3: $ST \leftarrow$ [`EmptyStack`] ▷ a new states stack (for reachable unexplored 'read' states)
4: $touched \leftarrow \emptyset$ ▷ a new states set (for states that were pushed to ST)
5: **for all** $ini \in I_s$ **do** ▷ find all reachable initial 'read' states
6: **for all** $succ \in S_s$ s.t. $succ$ is an eventual (read) successor of ini **do**
7: **if** $succ \notin touched$ **then** ▷ add a new state to I_a and S_a
8: push $succ$ to ST
9: $touched \leftarrow touched \cup \{succ\}$
10: $I_a \leftarrow I_a \cup \{succ\}$
11: $S_a \leftarrow S_a \cup \{succ\}$
12: $L_a(succ)|_{\{x\}} \leftarrow L_s(succ)|_{\{x\}}$, $L_a(succ)|_Y \leftarrow L_s(succ)|_{\tilde{Y}}$
13: **end if**
14: **end for**
15: **end for**
16: **while** $ST \neq$ [`EmptyStack`] **do** ▷ explore all reachable 'read' states
17: $st \leftarrow$ pop ST
18: $gen \leftarrow \{st\}$
19: **for** $i = 1, \ldots, m$ **do** ▷ find all m-th (last 'write') eventual successors of st
20: $nextgen \leftarrow \emptyset$ ▷ a new states set
21: **for all** $st_{gen} \in gen$ **do** ▷ find all i-th eventual successors of st
22: **for all** $succ \in S_s$ s.t. $succ$ is an eventual (write) successor of st_{gen} **do**
23: $nextgen \leftarrow nextgen \cup \{succ\}$
24: **end for**
25: **end for**
26: $gen \leftarrow nextgen$
27: **end for**
28: $nextgen \leftarrow \emptyset$ ▷ a new states set
29: **for all** $st_{gen} \in gen$ **do** ▷ find all 'eventual read successors' of st
30: **for all** $succ \in S_s$ s.t. $succ$ is an eventual (read) successor of st_{gen} **do**
31: $nextgen \leftarrow nextgen \cup \{succ\}$
32: **end for**
33: **end for**
34: **for all** $st_{ng} \in nextgen$ **do**
35: **if** $st_{ng} \notin touched$ **then** ▷ add a new state to S_a
36: push st_{ng} to ST
37: $touched \leftarrow touched \cup \{st_{ng}\}$
38: $S_a \leftarrow S_a \cup \{st_{ng}\}$
39: $L_a(st_{ng})|_{\{x\}} \leftarrow L_s(st_{ng})|_{\{x\}}$, $L_a(st_{ng})|_Y \leftarrow L_s(st_{ng})|_{\tilde{Y}}$
40: **end if**
41: $R_a \leftarrow R_a \cup \{(st, st_{ng})\}$ ▷ add a new transition to R_a
42: **end for**
43: **end while**
44: **return** i_a, I_a, L_a, S_a, R_a

Fig. 1. Algorithm for extracting T_a from T_s

In addition, we add to S_a new 'sink' states. For every $d \in D$, we add $sink_d$ such that $L_a(sink_d)|_Y$ is arbitrary and $L_a(sink_d)|_{\{x\}} = d$. For all $d_1, d_2 \in D$, $R_a(sink_{d_1}, sink_{d_2})$. We then add transitions to these sink states whenever for some input no transition is defined in S_a, ensuring that S_a is receptive. Formally, for every $d \in D$, if there exists no $s \in I_a$ such that $L_a(s)|_{\{x\}} = d$, we add $sink_d$ to I_a. For all $s \in S_a$ and $d \in D$, if there exists no $s' \in S_a$ such that $L_a(s')|_{\{x\}} = d$ and $R_a(s, s')$, then we add $R_a(s, sink_d)$. These additional states guarantee that T_a is receptive. In the case that T_s was receptive – as it should always be – all of these additions should be already taken care of simply by following the extraction algorithm. We describe them here only for the purpose of expressing that the extracted ILTS handles inputs that violate the environment's initial condition or safety component by continuing to a computation that would remain in sink states.

Constructively, the claim behind the following theorem is that, in some well-defined cased, the ILTS T_s that is extracted from the LTS T_a produces induced programs that asynchronously realize $\varphi(x; Y)$, as intended.

Theorem 9. *Let $\varphi(x; Y) = Imp(\varphi_e, \varphi_s)$, where $\varphi_e = \langle I_{\varphi_e}, S_{\varphi_e}, L_{\varphi_e} \rangle$, be a stutteringly robust specification with a GR(1) winning condition and with a memoryless environment, where $|Y| = \{y_1, \ldots, y_m\}$ and where there is exactly one input - x. Let r be a scheduling variable ranging over $\{1, \ldots, (1 + m)\}$, and let $\tilde{Y}$ be declared outputs variables.*

If $\psi(x, r; Y)$ is a stutteringly robust asynchronous strengthening of $\varphi(x; Y)$ such that $\mathcal{X}_{\psi}^{1,m}(x, r; Y \cup \tilde{Y})$ is synchronously realizable and where T_s is the (nondeterministic) LTS synthesized for it by the algorithm in [16], then the ILTS T_a, that is extracted from T_s, induces (after determinization) a program that asynchronously realizes $\varphi(x; Y)$.

Proof (Sketch): The algorithm takes a program T_s that realizes ψ and converts it to a program T_a. The program T_a 'jumps' from reading point to reading point in T_s. By using the declared outputs in $\tilde{Y}$ the asynchronous program does not have to commit on which reading point in T_s it moves to until the next input is actually read. By ψ being a strengthening of φ we get that the computation on T_s satisfies φ. Then, we use the stuttering robustness to make sure that the time that passes between reading points is not important for the satisfaction of φ. Memoryless-ness and single input are used to justify that prefixes of the computation on T_s can be extended with suffixes of other computations. Essentially, allowing us to 'copy-and-paste' segments of computations of T_s in order to construct one computation of T_a. □

Proof: In this proof we refer to the diagram from Fig. 2. In this diagram, all states have their variable assignments (labels) written on them, describing the values of the input x, all outputs $y_1, \ldots, y_n$, and the scheduler r. Only on some states we also write the values of the declared outputs $\tilde{Y}$, and to avoid clutter we simply write them separated from the rest of the variables (by a vertical line), as an additional value of outputs (without the '$\sim$' over them).

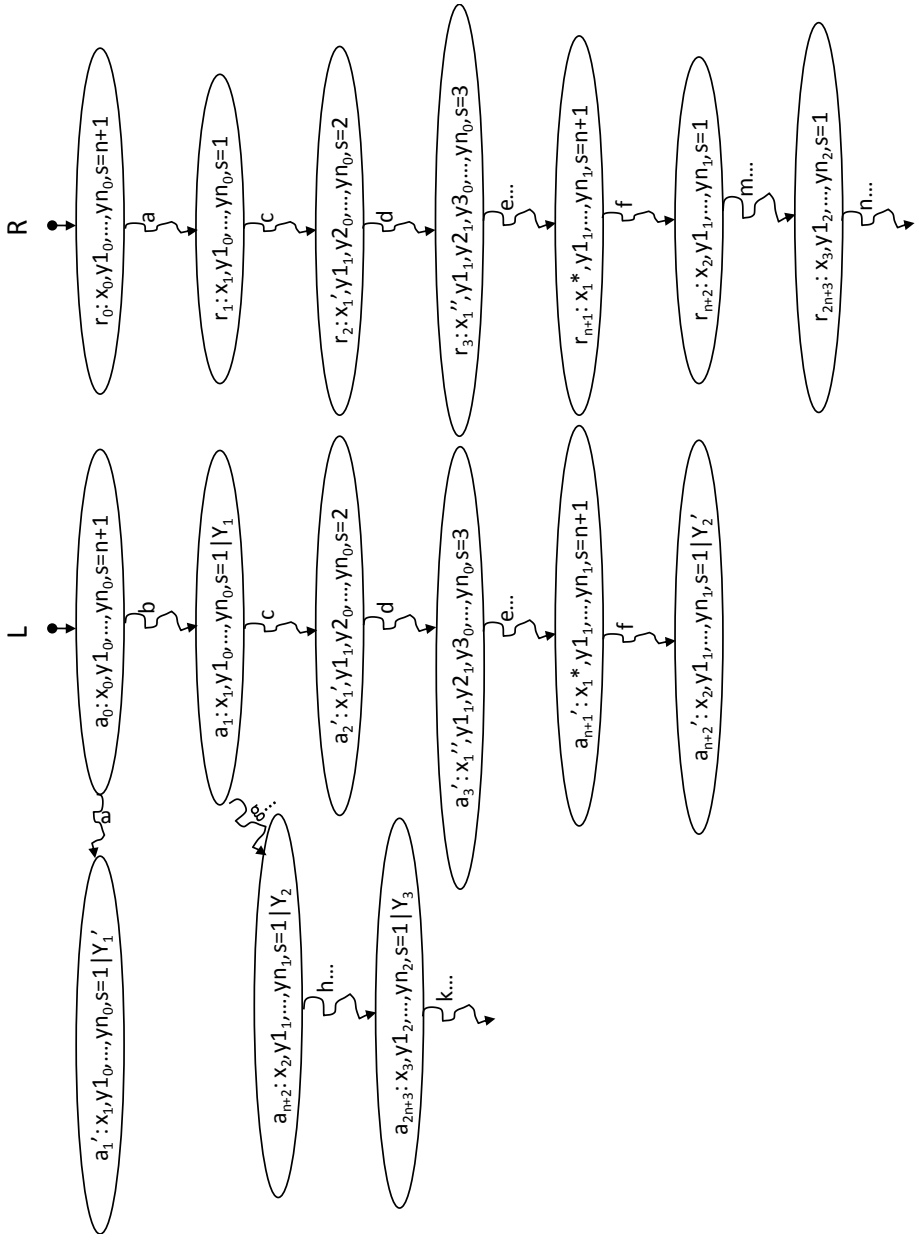

Fig. 2. The 'real' (R) and T_s (L) computations

We work with the nondeterministic LTS (ILTS) T_s (T_a), and show that any computation generated by any program that they could induce satisfies $\varphi(x;Y)$.

A *path* is a segment of a computation. We say that a path π is *safe for the environment*, if $\pi, 0 \models I_{\psi_e} \wedge \Box S_{\psi_e}$. (Since ψ is an asynchronous strengthening of φ, this is identical to saying that $\pi, 0 \models I_{\varphi_e} \wedge \Box S_{\varphi_e}$.)

Since $\varphi(x;Y)$ has a memory-less environment, and since this implies that $\psi(\{x,r\};Y)$ also has a memory-less environment, we allow ourselves to simply refer to, at times, a 'memory-less environment', where specifying the relevant specification is not critical.

The computation depicted on the right of Fig. 2, named R, with state names starting with the letter r, depicts the 'real' computation in the sense that it shows all of the transitions of both the system and the environment, guided by one program P_a induced by T_a (which considers, naturally, only inputs at reading points). On the left we describe a tree of paths guided by programs induced by T_s, named L, with state names starting with the letter a. L contains another computation that is generated by one program induced by T_s, that is also generated by the corresponding determinized T_a induced program P_a. Through the construction of the computation in L, we show that $R \models \varphi(x; y_1, \ldots, y_n)$.

Since we identify states in R by their assignment to variables, we freely 'borrow' the labeling function L_s to represent their value assignments. We also compare labeling of states from R and from L despite the fact that L labels cover $\tilde{Y}$ while R labels do not, referring only to the shared labels. Finally, we compare labeling of states by T_a and T_s, in the following way: For a state st, $L_a(st) = L_s(st)$ **iff** $L_a(st)|_Y = L_s(st)|_{\tilde{Y}}$ and $L_a(st)|_{\{x\}} = L_s(st)|_{\{x\}}$.

Let r^0, with $L_s(r^0) = \langle x^0, s^0, y_1^0, \ldots, y_n^0 \rangle$, also written as $L_s(r^0) = \langle x^0, s^0, Y^0 \rangle$ ($s^0 = n+1$), be the first state in R. If $L_s(r^0) \not\models I_e$, then $R, 0 \models \varphi(x;Y)$. Otherwise, due to the receptiveness of T_s, the state r^0 is also some initial state $a^0 \in I_s$ ($L_s(a^0) = L_s(r^0)$), and, by construction, $L_s(a^0) \models I_{\psi_e} \wedge I_{\psi_s}$. Also by construction, $i_a = Y^0$ (since we assume a single possible initial output, if $L_s(a^0) \not\models I_e$ then this must be the case), and T_a outputs a 'good' initial value. All paths in R and in L satisfy, therefore, $I_{\psi_e} \wedge I_{\psi_s}$.

If any prefix of R is not safe for the environment then, trivially, $R, 0 \models \varphi(x;Y)$. In the following, we assume, for completeness, that all prefixes of R are safe for the environment.

The path of R up until the first reading point, $r^0 \rightsquigarrow a$, exists in T_s - $a^0 \rightsquigarrow a$ (since T_s, by construction, induces programs that provide outputs against any behavior of the environment). Similarly, a state that agrees with the next state of R, r^1, on its labeling of $\{x,s\} \cup Y$ must be reachable (in one transition) from the last state of a in T_s - $a^0 \rightsquigarrow a \rightsquigarrow a^{1'}$ ($L_s(a^{1'})$ has some labeling for $\tilde{Y}$ - $\tilde{Y}^{1'} = \langle y_1^{1'}, \ldots, y_n^{1'} \rangle$). It is important to note here that both T_a and T_s might be non-deterministic. Let a^1 be some eventual successor of a^0 in T_s (through the path $a^0 \rightsquigarrow b \rightsquigarrow a^1$), that is also in T_a and that agrees with r^1 (and with $a^{1'}$) on the labeling of $\{x,s\} \cup Y$. By construction, both $a^{1'}$ and a^1 are in I_a, and T_a has 'good' initial states. We assume that, in this computation, a^1 was 'chosen'. Say that a^1 has the labeling of $\tilde{Y}$ $\tilde{Y}^1 = \langle y_1^1, \ldots, y_n^1 \rangle$. Let c be the path of R from r^1

up until the first writing point. Since the environment is memory-less (second assumption), and since r^1 and a^1 agree on the labeling of all the variable of S_{ψ_e}, the transition from a^1 to the first state of c must exist in T_s, and so are the rest of the transitions of c.

Let r^2 be the first writing state in R that immediately follows the last state of c, where the label of y_1 is changed from y_1^0 to y_1^1 (according to $\tilde{Y}^1$, as controlled by T_a through the selection of the eventual successor a^1 of a^0). Since (by assumption) the transition in R from the last state of c to r^2 is safe for the environment (note here that S_{ψ_e} is independent of Y' - the primed version of Y), than a transition from the last state of c in the path $a^0 \rightsquigarrow b \rightsquigarrow a^1 \rightsquigarrow c$ in T_s, to a state that agrees with r^2 on its labeling of x, must exist. Moreover, all eventual successors of a^1 in T_s must label the value y_1^1 to y_1. Let $a^{2'}$ be one such eventual successor of a^1 in T_s, that agrees with r^2 on its labeling of x (by the construction of T_a, that replaces its outputs labeling with the declared outputs labeling of T_s, it must exist in a path that is driven by T_a).

As R continues with a path that, starting from r^1, writes all outputs in writing points ($r^1 \rightsquigarrow c \rightsquigarrow r^2 \rightsquigarrow d \rightsquigarrow r^3 \rightsquigarrow e \ldots \rightsquigarrow r^{n+1}$), for similar arguments of memory-lessness of the environment and of agreement with the states of R on their labeling of $\{x, s\} \cup Y$, a similar path must exist in L - $a^1 \rightsquigarrow c \rightsquigarrow a^{2'} \rightsquigarrow d \rightsquigarrow a^{3'} \rightsquigarrow e \ldots \rightsquigarrow a^{(n+1)'}$. Moreover, the following path of R that originates in r^{n+1} and that continues up until, and including, the next reading point r^{n+2} - $r^{n+1} \rightsquigarrow r^{n+2}$, must also be duplicated in T_s - $a^{(n+1)'} \rightsquigarrow a^{(n+2)'}$ ($a^{(n+2)'}$ has some value $\tilde{Y}^{(n+2)'} = \langle y_1^{(n+2)'}, \ldots, y_n^{(n+2)'} \rangle$). All states on the path from r^1 to r^{n+1} agree with all states on the path from a^1 to $a^{(n+2)'}$, respectively, on their labels of $\{x, s\} \cup Y$ (including the 'writing' of the same output values at all writing points, and 'reading' of the same input value at the terminating reading point).

Since the state $a^{(n+2)'}$ exists in T_s then, by construction of T_a, at least one sequence of $n+1$ consecutive eventual successors from T_s that originate from a^1 (which is in T_a) would be represented in T_a through the value of $L_a(a^1)|_Y$, 'writing' in all writing points along the way all the values of $\tilde{Y}^1$ (one-by-one), and terminating in the state $a^{(n+2)}$ that agrees with $r^{(n+2)}$ on its labeling of $\{x, s\} \cup Y$. Say that $L_s(a^{(n+2)})|_{\tilde{Y}}$ is the labeling $\tilde{Y}^{(n+2)} = \langle y_1^{(n+2)}, \ldots, y_n^{(n+2)} \rangle$.

We showed that a path that corresponds to the writing-then-reading path $c \rightsquigarrow r^2 \rightsquigarrow d \rightsquigarrow r^3 \rightsquigarrow e \ldots \rightsquigarrow r^{n+1} \rightsquigarrow f \rightsquigarrow r^{(n+2)}$ of R exists in T_s and, therefore, represented in T_a. We continue inductively to construct a computation in T_s, called S_c, that is driven by one program induced by t_a - $a^0 \rightsquigarrow b \rightsquigarrow a^1 \rightsquigarrow g \ldots \rightsquigarrow a^{(n+2)} \rightsquigarrow h \ldots \rightsquigarrow a^{(2n+3)} \rightsquigarrow k \ldots$.

The key observation about S_c, other than the fact that it must exist (assuming that R is safe for the environment), is that it agrees with R on its interpretations of Y at all corresponding states, and that it agrees with R on its interpretations of x at all corresponding reading states. We know that the

following holds for S_c (using L_s, and as for all T_s computations)

$$S_c, 0 \models \begin{pmatrix} \alpha^{1,m}(s) & \wedge \\ I_{\psi_e} \wedge \Box\, S_{\psi_e} & \wedge \\ \psi(X \cup \{s\}; Y) & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_1(i) \wedge \neg \mathit{first}] \mathrel{\Rightarrow\!\!\!\!\!-} \mathit{unchanged}(y_i) \right] & \end{pmatrix}$$

Since ψ is an asynchronous strengthening of φ (specifically, due to the implication that is in that definition), we conclude that $S_c, 0 \models [\mathit{read}(1) \mathrel{\Rightarrow\!\!\!\!\!-} (x = \tilde{x})] \rightarrow \varphi(\tilde{x}; Y)$. Using the observation regarding the connection between R and S_c, we conclude that $R, 0 \models [\mathit{read}(1) \mathrel{\Rightarrow\!\!\!\!\!-} (x = \tilde{x})] \rightarrow \varphi(\tilde{x}; Y)$ and, in particular, that $R, 0 \models \varphi(x; Y)$. (Actually, in the transition from S_c to R we must account for the fact that the two computations may differ in length between every $I \backslash O$ point. To overcome that we use the stuttering robustness of φ, ψ and the rest of the clauses from the definitions of asynchronous strengthening.) $\square$

7 Applying the Realizability Test

We illustrate the application of the realizability test presented in Section 6. To come up with an asynchronous strengthening we propose the following heuristic.

Heuristic 1 *In order to derive an asynchronous strengthening $\psi(X \cup \{r\}; Y)$ for a specification $\varphi(X; Y)$, replace one or more occurrences of atomic formulae of inputs, e.g., $x_i = d$, by $(x_i = d) \wedge \ominus(r \neq i) \wedge (r = i)$, which means that $x_i = d$ at a reading point.*

The rationale here is to encode the essence of the stuttering quantification into the strengthening. Since this quantification requires indifference towards input values outside reading points, we state this explicitly.

We start with the 'response' specification $\varphi_3(x; y) = \mathit{Imp}(\varphi_{3,e}, \varphi_{3,s})$ from Section 5:

$$\varphi_3(x; y) \quad = \quad [\neg(x \leftrightarrow y) \mathrel{\Rightarrow\!\!\!\!\!-} (x \leftrightarrow \bigcirc x)] \rightarrow \begin{pmatrix} x \mathrel{\Rightarrow\!\!\!\!\!-} \diamondsuit\, y & \wedge \\ \overline{x} \mathrel{\Rightarrow\!\!\!\!\!-} \diamondsuit\, \overline{y} & \wedge \\ y \mathrel{\Rightarrow\!\!\!\!\!-} y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x & \wedge \\ \bigcirc(\overline{y} \mathrel{\Rightarrow\!\!\!\!\!-} \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x}) & \end{pmatrix}$$

This specification has a $GR(1)$ winning condition, it is stutteringly robust with a memory-less environment, and therefore it is potentially a good candidate to apply our heuristic. As suggested, we obtain the specification $\psi_3(x, r; y)$:

$$[\neg(x \leftrightarrow y) \mathrel{\Rightarrow\!\!\!\!\!-} (x \leftrightarrow \bigcirc x)] \rightarrow \begin{pmatrix} x \mathrel{\Rightarrow\!\!\!\!\!-} \diamondsuit\, y & \wedge \\ \overline{x} \mathrel{\Rightarrow\!\!\!\!\!-} \diamondsuit\, \overline{y} & \wedge \\ y \mathrel{\Rightarrow\!\!\!\!\!-} y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, [x \wedge \ominus(r = 2) \wedge (r = 1)] & \wedge \\ \bigcirc\{\overline{y} \mathrel{\Rightarrow\!\!\!\!\!-} \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, [\overline{x} \wedge \ominus(r = 2) \wedge (r = 1)]\} & \end{pmatrix}$$

We establish that ψ satisfies all our requirements. We then apply the synchronous realizability test of [16] to the kernel formula $\mathcal{X}_{\psi_3}(x, r; y)$. This formula is realizable and we get a LTS S_3 with 30 states and 90 transitions, which is then minimized, using a variant of the Myhill-Nerode minimization, to a LTS S'_3 with 16 states and 54 transitions. The algorithm in Fig. 1 constructs an ILTS $A_{S'_3}$ with 16 states and 54 transitions. Using model-checking [5] we ensure that all asynchronous interactions of $A_{S'_3}$ satisfy $\varphi_3(x; y)$. A simplified sub-ILTS of $A_{S'_3}$ that provides a complete strategy for $\varphi_3(x; y)$ is presented in Fig. 3 as an automaton. Notice, that this automaton has eight states, which, given that x, y, and r are Boolean variables, is the minimum possible.

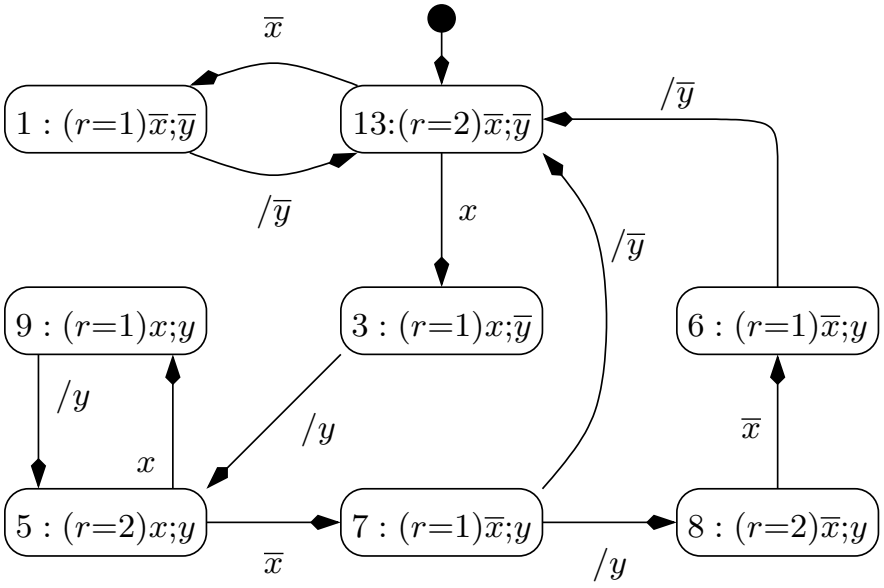

Fig. 3. ILTS (as an automaton)

Note that the automaton of Fig. 3 has a certain degree of nondeterminism as demonstrated in the exits out of state 7. At this point, it may nondeterministically choose to output y or $\overline{y}$. The automaton of Fig. 3 is a simplified version of the sub-automaton of $A_{S'_3}$. In particular, we have identified some states that have been found to have equivalent behavior. This led to the fact that the automaton does not contain the initial state 0 that has been identified with state 13 which is taken to be the initial state. Also, to avoid clutter, we omitted the representation of the *sink* state and all transitions entering it.

We devise similar specifications that copy the value of a Boolean input to one of several outputs according to the choice of the environment. Thus, we have a multi-valued input variable encoding the value and the target output variable

and several output variables. The specification $\varphi_4(x; y_0, y_1)$ is given below. The input x in $\varphi_2(x; y_0, y_1)$ ranges over $\{0, 1, 2, 3\}$. It's intended meaning is that $x = 0$ is interpreted as a 'request' to output $\overline{y_1}$, $x = 1$ to output y_1, $x = 2$ to output $\overline{y_0}$, and $x = 3$ to output y_0. The variable x actually implements a superposition of two inputs – one which selects an output to updates, and another that assigns a new value to it.

$$\varphi_{4,e}(x; y_0, y_1) = \begin{pmatrix} ((x = 0) \wedge y_1) \vee \\ ((x = 1) \wedge \overline{y_1}) \vee \\ ((x = 2) \wedge y_0) \vee \\ ((x = 3) \wedge \overline{y_0}) \end{pmatrix} \twoheadrightarrow \bigcirc \mathit{unchanged}(x)$$

$$\varphi_{4,s}(x; y_0, y_1) = \begin{pmatrix} (x = 0) \twoheadrightarrow \diamondsuit \overline{y_1} & \wedge \\ (x = 1) \twoheadrightarrow \diamondsuit y_1 & \wedge \\ (x = 2) \twoheadrightarrow \diamondsuit \overline{y_0} & \wedge \\ (x = 3) \twoheadrightarrow \diamondsuit y_0 & \wedge \\ y_0 \twoheadrightarrow y_0 \,\mathcal{S}\, \overline{y_0} \,\mathcal{S}\, (x = 3) & \wedge \\ y_1 \twoheadrightarrow y_1 \,\mathcal{S}\, \overline{y_1} \,\mathcal{S}\, (x = 1) & \wedge \\ \bigcirc[\overline{y_0} \twoheadrightarrow \overline{y_0} \,\mathcal{B}\, y_0 \,\mathcal{S}\, (x = 2)] & \wedge \\ \bigcirc[\overline{y_1} \twoheadrightarrow \overline{y_1} \,\mathcal{B}\, y_1 \,\mathcal{S}\, (x = 0)] & \end{pmatrix}$$

Using the same idea, we strengthen φ_4 to $\psi_4(x, r; y_0, y_1)$, which passes all the required tests. We then apply the synchronous realizability test in [16] to $\mathcal{X}_{\psi_4}(x, r; y_0, y_1)$ and get a LTS S_4 with 340 states and 1544 transitions, which is then minimized to 196 states and 1056 transitions. Our algorithm extracts an ILTS $A_{S_4'}$, which, as model checking confirms, asynchronously realizes φ_4.

From $\varphi_5(x; y_0, y_1, y_2)$ (similar to φ_4, with 3 outputs), we get a LTS with 1184 states and 8680 transitions.

8 Conclusions and Future Work

In this paper we extended the reduction of asynchronous synthesis to synchronous synthesis proposed in [19] to multiple input and output variables. We identify cases in which asynchronous synthesis can be done efficiently by bypassing the well known 'problematic' aspects of synthesis.

One of the drawbacks of this synthesis technique is the large size of resulting designs. However, we note that the size of asynchronous designs is bounded from above by synchronous designs. Thus, improvements to synchronous synthesis will result also in smaller asynchronous designs. We did not attempt to minimize or choose more effective synchronous programs, and we did not attempt to extract deterministic subsets of the nondeterministic controllers we worked with.

We believe that there is still room to explore additional cases in which asynchronous synthesis can be approximated. In particular, restrictions imposed by our heuristic (namely, one input environment and memory-less behavior) seem quite severe. Trying to remove some of these restrictions is left for future work.

Finally, asynchronous synthesis is related to solving games with partial information. There may be a connection between the cases in which synchronous

synthesis offers a solution to asynchronous synthesis and partial information games that can be solved efficiently.

Acknowledgments

We are very grateful to Lenore Zuck for helping with writing an earlier version of this manuscript.

References

1. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design Automation and Test in Europe*, pages 1188–1193, 2007.
2. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, volume 190 of *Electronic Notes in Computer Science*, pages 3–16, 2007.
3. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.
4. A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
6. D.C. Conner, H. Kress-Gazit, H. Choset, A. Rizzi, and G.J. Pappas. Valet parking without a valet. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 572–577. IEEE, 2007.
7. N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behavior models. In *18th International Symposium on Foundations of Software Engineering*, Santa Fe, NM, USA, 2010. ACM.
8. N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behavior models for fallible domains. In *33rd International Conference on Software Engineering*, Waikiki, HI, USA, May 2011. ACM.
9. T.A. Henzinger and N. Piterman. Solving games without determinization. In *Proc. 15th Annual Conf. of the European Association for Computer Science Logic*, volume 4207 of *Lect. Notes in Comp. Sci.*, pages 394–410. Springer-Verlag, 2006.
10. U. Klein, N. Piterman, and A. Pnueli. Effective Synthesis of Asynchronous Systems from GR(1) Specifications. In *Proc. 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lect. Notes in Comp. Sci.*, pages 283–298. Springer-Verlag, 2012.
11. U. Klein and A. Pnueli. Revisiting Synthesis of GR(1) Specifications. In *Hardware and Software: Verification and Testing (Proceedings of HVC'10)*, volume 6504 of *Lect. Notes in Comp. Sci.*, pages 161–181. Springer-Verlag, 2011.
12. H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3116–3121. IEEE, 2007.
13. H. Kugler, C. Plock, and A. Pnueli. Controller synthesis from lsc requirements. In *Proc. Fundamental Approaches to Software Engineering (FASE'09), volume 5503 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 79–93, 2009.

14. H. Kugler and I. Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Proc.* 15^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 5505 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 77–91, 2009.
15. O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. 46th IEEE Symp. Found. of Comp. Sci.*, 2005.
16. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006.
17. A. Pnueli and U. Klein. Synthesis of programs from temporal property specifications. In *Proc. 7th ACM/IEEE Intl. Conference on Formal Methods and Models for Codesign*, pages 1–7. IEEE Press, 2009.
18. 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.
19. 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.
20. A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV: A framework for developing verification algorithms. In *T. Touili, B. Cook, and P. Jackson, editors* Proc. 22^{nd} Intl. Conference on Computer Aided Verification (CAV'10), pages 171–174, 2010.
21. A. Pnueli and A. Zaks. On the merits of temporal testers. In *25 Years of Model Checking*, volume 5000 of *Lect. Notes in Comp. Sci.*, pages 172–195. Springer-Verlag, 2008.
22. M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.
23. Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *16th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lect. Notes in Comp. Sci.*, pages 127–142. Springer-Verlag, 2006.
24. A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with application to temporal logic. *Theor. Comp. Sci.*, 49:217–237, 1987.
25. M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *P. Wolper, editor,* Proc. 7^{th} Intl. Conference on Computer Aided Verification (CAV'95)*, volume 939 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 267–278, 1995.
26. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. and Cont.*, 115(1):1–37, 1994.
27. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning for dynamical systems. In *IEEE Conference on Decision and Control*, pages 5997–6004. IEEE press, 2009.
28. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Automatic synthesis of robust embedded control software. In *AAAI Spring Symposium on Embedded Reasoning: Intelligence in Embedded Systems*, 2010.
29. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon control for temporal logic specifications. In *Hybrid Systems: Computation and Control*, Lect. Notes in Comp. Sci. Springer-Verlag, 2010.