Effective Synthesis of Asynchronous Systems from GR(1) Specifications*

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

[1] Courant Institute of Mathematical Sciences, New York University
[2] Department of Computer Science, University of Leicester

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 [3] and subsequently solved by two techniques [2, 19]. In [15] 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 [15]. 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 [15].

This unfortunate situation led to extensive research on ways to bypass the complexity of synthesis (e.g., [11, 7, 13]). The work in [13] is of particular interest to us. It achieves scalability by restricting the type of handled specifications. This led to many applications of synthesis in various fields [1, 5, 24, 8, 10, 6]. 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 environ-

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

ment 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.

In [16], 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 [15], 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 [13]), the formula $\mathcal{X}$ is considerably more complex and requires the full treatment of [15].

Consequently, we propose a method to bypass this full reduction. In the invited paper [14] 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. It called for the construction of a weakening that could prove unrealizability through a simpler reduction to synchronous synthesis. This result is naturally extended to multiple variables, based on the extended Rosner reduction presented here, and is presented in an extended version [9]. In [14] we also outlined an approach 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 [14] 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 [14]). 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.

Proofs, which are omitted due to lack of space, are available in [9].

2 Preliminaries

Temporal Logic. We describe an extension of Quantified Propositional Temporal Logic (QPTL) [21] 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 usual standard abbreviations as well as: $(\forall^{\approx} x).\psi$ for $\neg(\exists^{\approx} x).(\neg\psi)$, $\psi_1 \mathcal{B} \psi_2$ for $\psi_1 \mathcal{S} \psi_2 \vee \boxminus \psi_1$, $\psi_1 \Rightarrow\!\!\!\!\!- \psi_2$ 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. 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 $squeeze(\sigma) = squeeze(\sigma')$.

Satisfaction of a QPTL formula φ over computation σ in location $i \geq 0$, denoted $\sigma, i \models \varphi$, is defined as usual. We define here only the case of quantification.

1. We have $\sigma, i \models (\exists x).\varphi$ **iff** $\sigma', i \models \varphi$ for some σ' that is an x-variant of σ.
2. We have $\sigma, i \models (\exists^{\approx} x).\varphi$ **iff** $\sigma'', i \models \varphi$ for some σ'' that is a x-variant of some stuttering variant σ' of σ.

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

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 $a \in D^X$ and $s \in S$ 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 $a \in D^X$ and every $s \in S$ 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 (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.

Theorem 1 ([15]). *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* $\varphi(X; Y)$*.*

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 [18]. 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 [13].

The Rosner Reduction. In [16], 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* $\mathcal{X}(x, r; y)$:

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

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.

Theorem 2 ([16]). *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 [15] 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*, that ranges over $\{1, \ldots, (n+m)\}$, which 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 $write_n(i)$ to indicate a state that is a writing point for y_i, $(r = i) \wedge \ominus(r \neq i)$ by $read(i)$ to indicate a state that is a reading point for x_i, $\bigwedge_{d \in D}[(z = d) \leftrightarrow \ominus(z = d)]$ by $unchanged(z)$ 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

$$\alpha^{n,m}(r) = \begin{pmatrix} r = (n+m) & \wedge \\ \bigwedge_{i=1}^{n+m} \Big[(r = i) \Rrightarrow \Big[(r = i)\, \mathcal{U} [r = (i \oplus_{n+m} 1)] \Big] \Big] & \end{pmatrix}$$

$$\beta^{n,m}(X \cup \{r\};Y) = \begin{pmatrix} \varphi(X;Y) & \wedge \\ \bigwedge_{i=1}^{m} \Big[[\neg write_n(i) \wedge \neg first] \Rrightarrow unchanged(y_i) \Big] & \wedge \\ (\forall^{\approx} \tilde{X}). \Big[\bigwedge_{i=1}^{n} [read(i) \Rrightarrow (x_i = \tilde{x_i})] \Big] \rightarrow \varphi(\tilde{X};Y) & \end{pmatrix}.$$

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 time 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 the possible asynchronous interactions. □

In principle, this theorem provides a complete solution to the problem of asynchronous synthesis (with multiple inputs and outputs). This requires to construct a deterministic automaton for $\mathcal{X}^{n,m}$ and to 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 [22]. Schewe and Finkbeiner extend these ideas to the case of branching time specifications [20]. 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 $Y = \{y_1, \ldots, y_m\}$.

A *by-demand program* P_b from X to Y is a function $P_b : D^* \mapsto \{1, \ldots, n\} \cup (D \times \{n+1, \ldots, n+m\})$. We 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, 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. Furthermore, 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 for every $1 \leq j \leq m$ we have $P_b(b_0[1], \ldots, b_0[j-1]) = (b_0[j], (n+j))$, and 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 $P_b(v(P_b, \sigma, c, i-1)) = (b_k[j], j)$.

Also, 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) and (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).

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.*

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

Fig. 1. Logical implication of asynchronous strengthening.

Proof (Sketch): A round-robin program is also 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 must satisfy the specification. □

5 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 [13] can be applied, bypassing much of the complexity involved in synthesis.

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 by 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* **asynchronous strengthening** *of $\varphi(X; Y)$ if $I_{\psi_e} = I_{\varphi_e}$, $S_{\psi_e} = S_{\varphi_e}$, and the implication in Fig. 1 is valid*

Checking the conditions in Def. 3 requires to check identity of propositional formulae and validity of a LTL formulae, which is supported, e.g., by JTLV [17].

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 [23], adding to it transitions that capture all stuttering options [16], 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 $\square \diamondsuit p$, where p is a Boolean formula. If the specification includes past temporal operators, these are embedded into the variables of the environment (c.f. [18]), 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 kernel formula defined in Fig. 2 *under-approximates* the original. The formula $declare^{n,m}$ 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.

Theorem 5. *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 memory-less 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 output variables.*

If $\psi(x, r; Y)$ *is a stutteringly robust asynchronous strengthening of* $\varphi(x; Y)$ *and* $\mathcal{X}_\psi^{1,m}(x, r; Y \cup \tilde{Y})$ *is synchronously realizable then* $\varphi(x; Y)$ *is asynchronously realizable. Furthermore, given a program that synchronously realizes* $\mathcal{X}_\psi^{1,m}$ *it can be converted in linear time to a program that asynchronously realizes* φ*.*

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. □

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 [13] 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.

$$
\begin{aligned}
\mathcal{X}_\psi^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) &= \alpha^{n,m}(r) \rightarrow \beta_\psi^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) \\
\beta_\psi^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) &= \begin{pmatrix} \mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y}) & \wedge \\ \psi(X \cup \{r\}; Y) & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \twoheadrightarrow \mathit{unchanged}(y_i) \right] & \end{pmatrix} \\
\mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y}) &= \\
& \left(\begin{array}{ll} \bigwedge_{i=1}^{m} [\mathit{write}_n(i) \twoheadrightarrow (y_i = \tilde{y}_i)] & \wedge \\ \left[\left[(r = \ominus r) \vee \bigvee_{i=1}^{m} [r = (n+i)] \right] \twoheadrightarrow \left[\bigwedge_{i=1}^{m} (\tilde{y}_i = \ominus \tilde{y}_i) \right] \right] & \end{array} \right).
\end{aligned}
$$

Fig. 2. The under approximation $\mathcal{X}_\psi^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$.

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. 3 *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.

As T_s is receptive, so is T_a. In particular the algorithm transfers sink states that handle violations of environment safety or initial conditions to T_a.

6 Applying the Realizability Test

We illustrate the application of the realizability test presented in Section 5. 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.

In [14] we showed how to strengthen the specification $\square(x \leftrightarrow y)$ to an asynchronously realizable specification with the same idea: a Boolean output y copies the value of an input x.

$$\varphi_1(x; y) \quad = \quad [\neg(x \leftrightarrow y) \Rightarrow\!\!\!\!\succ (x \leftrightarrow \bigcirc x)] \rightarrow \begin{pmatrix} x \Rightarrow\!\!\!\!\succ \diamondsuit y & \wedge \\ \overline{x} \Rightarrow\!\!\!\!\succ \diamondsuit \overline{y} & \wedge \\ y \Rightarrow\!\!\!\!\succ y \,\mathcal{S}\, \overline{y} \,\mathcal{S}\, x & \wedge \\ \bigcirc(\overline{y} \Rightarrow\!\!\!\!\succ \overline{y} \,\mathcal{B}\, y \,\mathcal{S}\, \overline{x}) & \end{pmatrix}$$

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 $\psi_1(x, r; y)$:

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

We establish that ψ satisfies all our requirements. We then apply the synchronous realizability test of [13] to the kernel formula $\mathcal{X}_{\psi_1}(x, r; y)$. This formula is realizable and we get a LTS S_1 with 30 states and 90 transitions, which is then minimized, using a variant of the Myhill-Nerode minimization, to a LTS S_1' with 16 states and 54 transitions. The algorithm in Fig. 3 constructs an ILTS $A_{S_1'}$ with 16 states and 54 transitions. Using model-checking [4] we ensure that all asynchronous interactions of $A_{S_1'}$ satisfy $\varphi_1(x; y)$.

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

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. 3. Algorithm for extracting T_a from T_s

a multi-valued input variable encoding the value and the target output variable and several outputs variables. The specification $\varphi_2(x; y_0, y_1)$ is given below.

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

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

Using the same idea, we strengthen φ_2 to $\psi_2(x, r; y_0, y_1)$, which passes all the required tests. We then apply the synchronous realizability test in [13] to $\mathcal{X}_{\psi_2}(x, r; y_0, y_1)$ and get a LTS S_2 with 340 states and 1544 transitions, which is then minimized to 196 states and 1056 transitions. Our algorithm extracts an ILTS $A_{S_2'}$, which, as model checking confirms, asynchronously realizes φ_2.

From $\varphi_3(x; y_0, y_1, y_2)$ (similar to φ_2, with 3 outputs), we get a LTS with 1184 states and 8680 transitions.

7 Conclusions and Future Work

In this paper we extended the reduction of asynchronous synthesis to synchronous synthesis proposed in [16] 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 L. Zuck for helping 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 *DATE*, pages 1188–1193, 2007.
2. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
3. A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.
4. E.C. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
5. D.C. Conner, H. Kress-Gazit, H. Choset, A. Rizzi, and G.J. Pappas. Valet parking without a valet. In *IRSES*, pages 572–577. IEEE, 2007.
6. N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behavior models for fallible domains. In *ICSE*, 2011. ACM.
7. T.A. Henzinger and N. Piterman. Solving games without determinization. In *CSL*, LNCS 4207, pages 394–410. Springer-Verlag, 2006.
8. H. Kugler, C. Plock, and A. Pnueli. Controller synthesis from lsc requirements. In *FASE*, LNCS 5503, pages 79–93. Springer-Verlag, 2009.
9. U. Klein, N. Piterman, and A. Pnueli. Effective Synthesis of Asynchronous Systems from GR(1) Specifications. Tech Rep TR2011-944, Courant Inst of Math Sci, NYU.
10. H. Kugler and I. Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *TACAS*, LNCS 5505, pages 77–91. Springer-Verlag, 2009.
11. O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *FOCS*, 2005.
12. N. Piterman and A. Pnueli. Faster solution of Rabin and Streett games. In *LICS*. IEEE, IEEE press, 2006.
13. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, LNCS 3855, pages 364–380. Springer-Verlag, 2006.
14. A. Pnueli and U. Klein. Synthesis of programs from temporal property specifications. In *MEMOCODE*, pages 1–7. IEEE Press, 2009.
15. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
16. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, LNCS 372, pages 652–671. Springer-Verlag, 1989.
17. A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV: A framework for developing verification algorithms. In *CAV*, LNCS 6174, pages 171–174. Springer-Verlag, 2010.
18. A. Pnueli and A. Zaks. On the merits of temporal testers. In *25 Years of Model Checking*, LNCS 5000, pages 172–195. Springer-Verlag, 2008.
19. M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, AMS, 1972.
20. S. Schewe and B. Finkbeiner. Synthesis of Asynchronous Systems. In *LOPSTR*, LNCS 4407, pages 127–142. Springer-Verlag, 2006.
21. A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi autamata with application to temporal logic. *TCS*, 49:217–237, 1987.
22. M.Y. Vardi An Automata-Theoretic Approach to Fair Realizability and Synthesis In *CAV*, LNCS 939, pages 267–278. Springer-Verlag, 1995.
23. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *I&C*, 115(1):1–37, 1994.
24. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon control for temporal logic specifications. In *HSCC*, LNCS. Springer-Verlag, 2010.