Synthesis Of Asynchronous Reactive Programs From Temporal Specifications

Suguman Bansal[1], Kedar S. Namjoshi[2], and Yaniv Sa'ar[3]

[1] Rice University, Houston, TX, USA, suguman@rice.edu
[2] Bell Labs, Nokia, Murray Hill, NJ, USA, kedar.namjoshi@nokia-bell-labs.com
[3] Bell Labs, Nokia, Kfar Saba, Israel, yaniv.saar@nokia.bell-labs.com

Abstract. Asynchronous interactions are ubiquitous in computing systems and complicate design and programming. Automatic construction of asynchronous programs from specifications ("synthesis") could ease the difficulty, but known methods are complex, and intractable in practice. This work develops substantially simpler synthesis methods. A direct, exponentially more compact automaton construction is formulated for the reduction of asynchronous to synchronous synthesis. Experiments with a prototype implementation of the new method demonstrate practical feasibility. Furthermore, it is shown that for several useful classes of temporal properties, automaton-based methods can be avoided altogether and replaced with simpler Boolean constraint solving.

1 Introduction

Modern software and hardware systems harness asynchronous interactions to improve speed, responsiveness, and power consumption: delay-insensitive circuits, networks of sensors, multi-threaded programs and interacting web services are all asynchronous in nature. Various factors contribute to asynchrony, such as unpredictable transmission delays, concurrency, distributed execution, and parallelism. The common result is that each component of a system operates with partial, out-of-date knowledge of the state of the others, which considerably complicates system design and programming. Yet, it is often easier to state the desired behavior of an asynchronous program. We therefore consider the question of automatically constructing (i.e., synthesizing) a correct reactive asynchronous program directly from its temporal specification.

The *asynchronous synthesis problem* was originally formulated by Pnueli and Rosner in 1989 on the heels of their work on synchronous synthesis [31,32]. The task is that of constructing a (finite-state) program which interacts asynchronously with its environment while meeting a temporal specification on the actions at the interface between program and environment. Given a linear temporal specification φ, Pnueli-Rosner show that *asynchronous* synthesis can be reduced to checking whether a derived specification φ', specifying the required behavior of the scheduler, is *synchronously* synthesizable. That is, an asynchronous program can implement φ iff a synchronous program can implement φ'.

It may then appear straightforward to construct asynchronous programs using one of the many tools that exist for synchronous synthesis. However, the derived formula φ' embeds a nontrivial stutter quantification, which requires a complex intermediate automaton construction; it has not, to the authors' knowledge, ever been implemented. This situation is in stark contrast to that of synchronous synthesis, for which multiple tools and algorithms have been created.

Alternative methods have been proposed for asynchronous synthesis: Finkbeiner and Schewe reduce a bounded form of the problem to a SAT/SMT query [35], and Klein, Piterman and Pnueli show that some GR(1) specifications[4] can be transformed as above to an approximate synchronous GR(1) property [21,22]. These alternatives, however, have drawbacks of their own. The SAT/SMT reduction is exponential in the number of interface (input and output) bits, an important parameter; the GR(1) specifications amenable to transformation are limited and are characterized by semantic conditions that are not easily checked.

This work presents two key simplifications. First, we define a new property, $\mathsf{PR}(\varphi)$ (named in honor of Pnueli-Rosner's pioneering work) which, like φ', is synchronously realizable if, and only if, φ is asynchronously realizable. We then present an automaton construction for $\mathsf{PR}(\varphi)$ that is direct and simpler, and results in an exponentially smaller automaton than the one for φ'. In particular, the automaton for $\mathsf{PR}(\varphi)$ has only at most *twice* the states of the automaton for φ, as opposed to the *exponential blowup* of the state space (in the number of interface bits) incurred in the construction of the automaton for φ'. As almost all synchronous automaton-based synthesis tools use an explicit encoding for automaton states, this reduction is vital in practice.

We show how to implement the transformation PR symbolically (with BDDs), so that interface bits are always represented in symbolic form. One can then apply the modular strategy of Pnueli-Rosner: a symbolic automaton for φ is transformed to a symbolic automaton for $\mathsf{PR}(\varphi)$ (instead of φ'), which is analyzed with a synchronous synthesis tool. We establish that PR is conjunctive and preserves safety[5]. These are important properties, used by tools such as Acacia+ [8] and Unbeast [11] to optimize the synchronous synthesis task. The new construction has been implemented in a prototype tool, BAS, and experiments demonstrate feasibility in practice.

In addition, we establish that for several classes of temporal properties, which are easily characterized by syntax, the automaton-based method can be avoided entirely and replaced with Boolean constraint solving. The constraints are quantified Boolean formulae, with prefix $\exists\forall$ and a kernel that is derived from the original specification. This surprising reduction, which resolves a temporal problem with Boolean reasoning, is a consequence of the highly adversarial role of the environment in the asynchronous setting.

These contributions turn a seemingly intractable synthesis task into one that is feasible in practice.

---

[4] The GR(1) ("General Reactivity (1)") subclass has an efficient symbolic procedure for synchronous synthesis, formulated in [28] and implemented in several tools.

[5] I.e., $\mathsf{PR}(\bigwedge_i f_i) = \bigwedge_i \mathsf{PR}(f_i)$, and $\mathsf{PR}(f)$ is a safety property if f is a safety property.

2 Preliminaries

Temporal Specifications Linear Temporal Logic (LTL) [29] extends propositional logic with temporal operators. LTL formulae are defined as $\varphi ::= \mathsf{True} \mid \mathsf{False} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2 \mid \Diamond\varphi \mid \Box\varphi \mid \boxminus\varphi$. Here p is a proposition, and X(Next), U (Until), $\Diamond$ (Eventually), $\Box$ (Always), and $\boxminus$(Always in the past) are temporal operators. The LTL semantics is standard, and is in the full version of the paper. For an LTL formula φ, let $\mathcal{L}(\varphi)$ denote the set of words (over subsets of propositions) that satisfy φ.

GR(1) is a useful fragment of LTL, where formulae are of the form $(\Box S_e \wedge \bigwedge_{i=0}^{m} \Box\Diamond P_i) \Rightarrow (\Box S_s \wedge \bigwedge_{i=0}^{n} \Box\Diamond Q_i)$, for propositional formulae S_e, S_s, P_i, Q_i. Typically, the left-hand side of the implication is used to restrict the environment, by requiring safety and liveness assumptions to hold, while the right-hand side is used to define the safety and liveness guarantees required of the system.

LTL specifications can be turned into equivalent Büchi automata, using standard constructions. A Büchi automaton, A, is specified by the tuple $(Q, Q_0, \Sigma, \delta, G)$, where Q is a set of states, $Q_0 \subseteq Q$ defines the initial states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $G \subseteq Q$ defines the "green" (also known as "accepting" or "final") states. A *run* r of the automaton on an infinite word $\sigma = a_0, a_1, \ldots$ over Σ is an infinite sequence $r = q_0, a_0, q_1, a_1, \ldots$ such that q_0 is an initial state, and for each k, (q_k, a_k, q_{k+1}) is in the transition relation. Run r is accepting if a green state appears on it infinitely often; the language of A, denoted $\mathcal{L}(A)$, is the set of words that have an accepting run.

The Asynchronous Synthesis Model The goal of synthesis is to construct an "open" program M meeting a specification at its interface. In the asynchronous setting, the program M interacts in a fair interleaved manner with its environment E. The fairness restriction requires that E and M are each scheduled infinitely often in all infinite executions. Let $E//M$ denote this composition. The interface between E and M is formed by the variables x and y. Variable x is written by E and is read-only for M, while y is written by M and is read-only for E. One can consider x (resp., y) to represent a vector of variables, i.e., $x = (x_1, \ldots, x_n)$ (resp., $y = (y_1, \ldots, y_m)$) which is read (resp., written) atomically. Many of our results also extend to non-atomic reads and writes, and are discussed in the full version of the paper.

The synthesis task is to construct a program M which satisfies a temporal property $\varphi(x, y)$ over the interface variables in the composition $E//M$, for *any* environment E. The most adversarial environment is the one which sets x to an arbitrary value at each scheduled step, we denote it by $\mathsf{CHAOS}(x)$. The behaviors of the composition $\mathsf{CHAOS}(x)//M$ simulate those of $E//M$ for all E. Hence, it suffices to produce M which satisfies φ in the composition $\mathsf{CHAOS}(x)//M$. One can limit the set of environments through an assumption in the specification.

This leads to the formal definition of an *asynchronous schedule*, given by a pair of functions, $r, w : \mathbb{N} \to \mathbb{N}$, which represent read and write points, respectively. The initial write point, $w(0) = 0$, and represents the choice of initial value

for the variable y. Without loss of generality, the read-write points alternate, i.e., for all $i \geq 0$, $w(i) \leq r(i) < w(i+1)$ and $r(i) < w(i+1) \leq r(i+1)$. A *strict* asynchronous schedule does not allow read and write points to overlap, i.e., the constraints are strengthened to $w(i) < r(i) < w(i+1)$ and $r(i) < w(i+1) < r(i+1)$. A *tight* asynchronous schedule is the strict schedule without any non-read-write gaps, i.e., $r(k) = 2k+1$ and $w(k) = 2k$, for all k. A *synchronous* schedule is the special non-strict schedule where $r(i) = i$ and $w(i) = i$, for all i.

Let D^v denote the binary domain $\{\mathsf{True}, \mathsf{False}\}$ for a variable v. A program M can be represented semantically as a function $f : (D^x)^* \rightarrow D^y$. For an asynchronous schedule (r, w), a sequence $\sigma = (D^x \times D^y)^\omega$ is said to be an *asynchronous execution of f over (r, w)* if the value of y is changed only at writing points, in a manner that depends only on the values of x at prior reading points. Formally, for all $i \geq 0$, $y_{w(i+1)} = f(x_{r(0)} \ldots x_{r(i)})$, and for all j such that $w(i) \leq j < w(i+1)$, $y_j = y_{w(i)}$. The initial value of y is the value it has at point $w(0) = 0$. The set of such sequences is denoted as $\mathsf{asynch}(f)$. Over synchronous schedules, the set of such sequences is denoted by $\mathsf{synch}(f)$. Function f is an asynchronous implementation of φ if all asynchronous executions of f over all possible schedules satisfy φ, i.e., if $\mathsf{asynch}(f) \subseteq \mathcal{L}(\varphi)$.

This formulation agrees with that given by Pnueli and Rosner for strict schedules. For synchronous schedules (and other non-strict schedules), our formulation has a Moore-style semantics – the output depends on strictly earlier inputs – while Pnueli and Rosner formulate a Mealy semantics. A Moore semantics is more appropriate for modeling software programs, where the output variable is part of the state, and fits well with the theoretical constructions that follow.

Definition 1 (Asynchronous LTL Realizability). *Given an LTL property $\varphi(x, y)$ over the input variable x and output variable y, the* asynchronous LTL realizability *problem is to determine whether there is an asynchronous implementation for φ.*

Definition 2 (Asynchronous LTL Synthesis). *Given a realizable LTL-formula φ, the* asynchronous LTL synthesis *problem is to construct an asynchronous implementation of φ.*

Examples Pnueli and Rosner give a number of interesting specifications. The specification $\Box\,(y \equiv Xx)$ ("the current output equals the next input") is satisfiable but not realizable, as any implementation would have to be clairvoyant. On the other hand, the flipped specification $\Box\,(x \equiv Xy)$ ("the next output equals the current input") is synchronously realizable by a Moore machine which replays the current input as the next output. The specification $\Diamond\Box\, x \equiv \Diamond\Box\, y$ is synchronously realizable by the same machine, but is asynchronously unrealizable, as shown next. Consider two input (x) sequences, under a schedule where reads happen only at odd positions. In both, let x=true at all reading points. Then any program must respond to both inputs with the same output sequence for y. Now suppose that in the first sequence x is false at all non-read positions, while in the second, x is true at all non-reading positions. In the first case, the specification

forces the output y-sequence to be false infinitely often; in the second, y is forced to be true from some point on, a contradiction.

The negated specification $\Diamond\Box x \not\equiv \Diamond\Box y$ is also asynchronously unrealizable, for the same reason. This "gap" illustrates an intriguing difference from the synchronous case, where either a specification is realizable for the system, or its negation is realizable for the environment. The two halves of the equivalence, i.e., $\Diamond\Box x \Rightarrow \Diamond\Box y$ and $\Diamond\Box y \Rightarrow \Diamond\Box x$ are individually asynchronously realizable, by strategies that fix the output to y=true and to y=false, respectively.

From Asynchronous to Synchronous Synthesis Pnueli and Rosner reduced asynchronous LTL synthesis to synchronous synthesis of Büchi objectives. Their reduction applied to LTL-formula with a single input and output variable [32]; it was later extended to the non-atomic case [30]. The original Rosner-Pnueli reduction deals exclusively with strict schedules, since they showed that it is sufficient to consider only strict schedules.

Two infinite sequences are said to be *stuttering equivalent* if one sequence can be obtained from the other by a finite duplication ("stretching") of a given state or by deletion ("compressing") of finitely many contiguous identical states retaining at least one of them. The *stuttering quantification* $\exists^{\approx}$ is defined as follows: $\exists^{\approx} x.\varphi$ holds for sequence π if $\exists x.\varphi$ holds for a sequence π' that is stuttering equivalent to π. Pnueli-Rosner showed that LTL-formula $\varphi(x,y)$ over input x and output y is asynchronously realizable iff a "kernel" formula (this is the precise formula referred to as φ' in the Introduction) $\mathcal{K}(r,w,x,y) = \alpha(r,w) \rightarrow \beta(r,w,x,y)$ over read sequence r, write sequence w, input sequence x and output sequence y is synchronously realizable:

$$\begin{aligned}
\alpha(r,w) = \quad & (\neg r \wedge \neg w \,\mathsf{U}\, r) \wedge \Box\neg(r \wedge w) \wedge \Box\,(r \;\Rightarrow\; (r \,\mathsf{U}\, (\neg r) \,\mathsf{U}\, w)) \\
& \wedge \Box\,(w \;\Rightarrow\; (w \,\mathsf{U}\, (\neg w) \,\mathsf{U}\, r)) \\
\beta(r,w,x,y) = \quad & \varphi(x,y) \wedge \forall a.\Box\,((y=a) \;\Rightarrow\; ((y=a) \,\mathsf{U}\, (\neg w \wedge (y=a) \,\mathsf{U}\, w))) \\
& \wedge \forall^{\approx} x'.(\Box\,(\neg r \;\Rightarrow\; \neg r \,\mathsf{U}\, (x=x')) \;\Rightarrow\; \varphi(x',y))
\end{aligned}$$

Here, α encodes the strict scheduling constraints on read and write points, while β encodes conditions which assure a correct asynchronous execution over (r,w). The $\forall^{\approx}$ quantification, intuitively, quantifies over all adversarial schedules similar to the current (r,w): it requires φ to hold over all sequences obtained from the current sequence σ by stretching or compressing the segments between read and write points, and choosing different values for x on those segments.

3 Symbolic Asynchronous Synthesis

Pnueli and Rosner's procedure for asynchronous synthesis [32] is as follows: first, a Büchi automaton is built for the kernel formula $\neg\mathcal{K}$. This automaton is then determinized and complemented to form a deterministic word automaton for $\mathcal{K}$, which is then re-interpreted as a tree automaton and tested for non-emptiness. The transformations use standard constructions, except for the interpretation

of the $\exists^{\approx}$ operator in the formation of the Büchi automaton for $\neg\mathcal{K}$. For a Büchi automaton A, an automaton for $\exists^{\approx}\mathcal{L}(A)$ is constructed in two steps: first applying a "stretching" transformation on A, followed by a "compressing" transformation. Stretching introduces new automaton states of the form (q, a), for each state q of A and each letter a.

When this general construction is applied to the formula $\neg\mathcal{K}$, the alphabet of the automaton A is formed of all possible valuations of the pair of variables (x, y), which has size *exponential* in the number of interface bits. The stretching step introduces a copy of an automaton state for each letter, which results in an exponential blow-up of the state space of the constructed automaton. As all current tools for synchronous synthesis represent automaton states explicitly[6], the exponential blowup introduced by the stuttering quantification is a significant obstacle to implementation.

In Pnueli-Rosner's construction, the determinization and complementation steps are also complex, utilizing Safra's construction. These steps are simplified by the "Safraless" procedure adopted in current tools for synchronous synthesis.

The other major issue with the Pnueli-Rosner construction is that the kernel formula $\mathcal{K}$ introduces the scheduling variables r, w as input variables. However, the actions of a synthesized program should not rely on the values of these variables. Pnueli-Rosner ensure this by checking satisfiability over "canonical" tree models; it is unclear, however, how to realize this effect using a synchronous synthesis tool as a black box.

We define a new property, $\mathsf{PR}(\varphi)$, that differs from $\mathcal{K}$ but, similarly, is synchronously realizable if, and only if, φ is asynchronously realizable. We then present an automaton construction for $\mathsf{PR}(\varphi)$ that bypasses the general construction for $\exists^{\approx}$, avoiding the exponential blowup and resulting in an automaton with *at most twice* the states of the original. Moreover, this construction refers only to x and y, avoiding the second issue as well. We then show that this construction can be implemented fully symbolically.

3.1 Basic Formulations and Properties

As formulated in Section 2, an asynchronous execution of f is determined by the schedule (r, w). For a strict schedule, any infinite sequence representing an asynchronous behavior of f over (r, w) may be partitioned into a sequence of *blocks*, as follows. The start of the i'th block is at the i'th writing point, $w(i)$, and ends just before the $i+1$'st writing point, $w(i+1)$. The schedule ensures the i'th block includes the i'th reading point, $r(i)$, associated with the input-output value (x_i, y_i). As the value of y changes only at writing points, y_i is constant in the i'th block. Thus, the i'th block follows the pattern $(\bot, y_i)^*(x_i, y_i)(\bot, y_i)^*$, where $\bot$ denotes an arbitrary choice of x-value. Figure 1 illustrates a strict asynchronous computation and its decomposition into blocks.

[6] With one exception. BoSy's DQBF procedure is fully symbolic but does not work as well as the default QBF procedure [12].

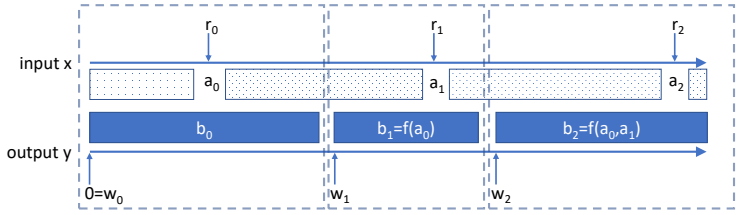

Fig. 1. A strict asynchronous computation for f. Values of x at non-reading points are shown as dotted. The y-value is constant between writing points, illustrated by a solid rectangle. Blocks are shown as dashed rectangles.

Expansions. The set of *expansions* of sequence $\delta = (x_0, y_0)(x_1, y_1)\ldots$ consists of all sequences obtained by simultaneously replacing each (x_i, y_i) in δ by a block with the pattern $(\bot, y_i)^*(x_i, y_i)(\bot, y_i)^*$. Formally, given sequences $\delta = (x_0, y_0)(x_1, y_1)\ldots$ and $\sigma = (\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1)\ldots$, δ *expands to* σ, denoted as $\delta\, \mathsf{exp}\, \sigma$, if there exists an asynchronous schedule $(\hat{r}, \hat{w})$ for which σ is an execution that is a block pattern of δ, i.e., for all i, $x_i = \bar{x}_{\hat{r}(i)}$ and $y_i = \bar{y}_{\hat{w}(i)}$ and for all j, $\hat{w}(i) \leq j < \hat{w}(i+1)$ it is the case that $\bar{y}_j = \bar{y}_{\hat{w}(i)}$. The inverse relation (read as *contracts to*) is denoted by exp^{-1}. Figure 2 shows the synchronous computation that contracts the computation shown in Figure 1.

Relational Operators. For a relation R, the modal operators $\langle R\rangle$ and $[R]$ are defined as follows. For any set S,

$$u \in \langle R\rangle S = (\exists v : uRv \wedge v \in S) \qquad u \in [R]S = (\forall v : uRv \Rightarrow v \in S)$$

By definition, the operators are negation duals, i.e., $\neg\langle R\rangle(\neg S) = [R](S)$ for any R and any S. For an LTL formula φ and a relation R over infinite sequences, we let $\langle R\rangle\varphi$ abbreviate $\langle R\rangle(\mathcal{L}(\varphi))$, and similarly, let $[R]\varphi$ abbreviate $[R](\mathcal{L}(\varphi))$.

Galois Connections. Given partial orders $(A, \preceq_A)$ and $(B, \preceq_B)$, a pair of functions $g : A \to B$ and $h : B \to A$ form a Galois connection if, for all $a \in A, b \in B$: $g(a) \preceq_B b$ is equivalent to $a \preceq_A h(b)$. From the definitions, it is clear that the operators $(\langle R^{-1}\rangle, [R])$ form a Galois connection over the partial orders defined by the subset relation. I.e., for any sets S and T: $\langle R^{-1}\rangle S \subseteq T$ iff, $S \subseteq [R]T$.

We first establish that the asynchronous executions of f are precisely the synchronous executions of f under an inverse expansion.

Theorem 1. *For an implementation* f, $\mathsf{asynch}(f) = \langle\, \mathsf{exp}^{-1}\, \rangle \mathsf{synch}(f)$.

Proof. (ping) Let σ be an execution in $\mathsf{asynch}(f)$, generated for some schedule (r, w). For any k, consider the k'th block of σ. This is the set of positions from $w(k)$ to $w(k+1) - 1$, which includes the k'th reading point $r(k)$, say with the value (x_k, y_k). Then the block follows the pattern $(\bot, y_k)^*(x_k, y_k)(\bot, y_k)^*$. So σ is an expansion of the sequence $\delta = (x_0, y_0)(x_1, y_1)\ldots$. By the definition of an asynchronous execution, the value $y_{k+1} = f(x_0, \ldots, x_k)$. This is precisely the

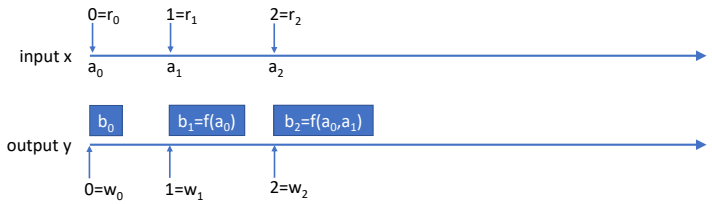

Fig. 2. The contracted synchronous (Moore) computation

requirement for δ to be a synchronous execution of f. Hence, we have that there is a δ such that $\delta \,\mathsf{exp}\, \sigma$ and $\delta \in \mathsf{synch}(f)$. Therefore, $\sigma \in \langle\, \mathsf{exp}^{-1} \,\rangle \mathsf{synch}(f)$.

(pong) Let σ be in $\langle\, \mathsf{exp}^{-1} \,\rangle \mathsf{synch}(f)$. By definition, there is a $\mathsf{synch}(f)$ execution $\delta = (x_0, y_0)(x_1, y_1)\ldots$ such that $\delta \,\mathsf{exp}\, \sigma$. As δ is a synchronous execution of f, the value $y_{k+1} = f(x_0, x_1, \ldots, x_k)$, for all k. Then σ is an asynchronous execution of f under the schedule where the k-th reading point is the point that the k'th entry, (x_k, y_k), from δ is mapped to in σ, and the $(k+1)$-th writing point is the first point of the $(k+1)$'st block in the expansion. □

We now use the Galois connection to show how asynchronous synthesis can be reduced to an equivalent synchronous synthesis task. Consider a property φ that must hold asynchronously for an implementation f.

Theorem 2. *Let f be an implementation function, and φ a property. Then* $\mathsf{asynch}(f) \subseteq \mathcal{L}(\varphi)$ *if, and only if,* $\mathsf{synch}(f) \subseteq [\,\mathsf{exp}\,]\varphi$.

Proof. From Theorem 1, $\mathsf{asynch}(f) \subseteq \mathcal{L}(\varphi)$ holds iff $\langle\, \mathsf{exp}^{-1} \,\rangle \mathsf{synch}(f) \subseteq \mathcal{L}(\varphi)$ does. By the Galois connection, this is equivalent to $\mathsf{synch}(f) \subseteq [\,\mathsf{exp}\,]\varphi$. □

3.2 The Pnueli-Rosner Closure

We refer to the property $[\,\mathsf{exp}\,]\varphi$ as the Pnueli-Rosner closure of φ, in honor of their pioneering work on this problem, and denote it by $\mathsf{PR}(\varphi)$. This has interesting mathematical properties, which are useful in practice.

Theorem 3. $\mathsf{PR}(\varphi) = [\,\mathsf{exp}\,]\varphi$ *has the following properties.*

1. *(Closure)* PR *is monotonic and a downward closure, i.e.,* $\mathsf{PR}(\varphi) \subseteq \mathcal{L}(\varphi)$
2. *(Conjunctivity)* PR *is conjunctive, i.e.,* $\mathsf{PR}(\bigwedge_i \varphi_i) = \bigcap_i \mathsf{PR}(\varphi_i)$
3. *(Safety Preservation) If φ is a safety property, so is* $\mathsf{PR}(\varphi)$

The closure property relies on the reflexivity and transitivity of exp, and that $[R]$ is monotonic for every R. Conjunctivity follows from the conjunctivity of $[R]$ for any R. Safety preservation is based on the Alpern-Schneider [4] formulation of safety over infinite words. Proofs are in the full version of the paper.

Conjunctivity is exploited by the tools Acacia+ [8] and Unbeast [11] to optimize the synchronous synthesis procedure. The Unbeast tool also separates out

safety from non-safety sub-properties to optimize the synthesis procedure. Thus, if a specification φ has the form $\varphi_1 \wedge \varphi_2$, where φ_1 is a safety property, then $\mathsf{PR}(\varphi) = \mathsf{PR}(\varphi_1) \cap \mathsf{PR}(\varphi_2)$ also denotes the intersection of the safety property $\mathsf{PR}(\varphi_1)$ with another property.

3.3 The Closure Automaton Construction

By negation duality, $\mathsf{PR}(\varphi)$ equals $\neg\langle\,\mathsf{exp}\,\rangle(\neg\varphi)$. We use this property to reduce asynchronous to synchronous synthesis, as follows.

1. Construct a non-deterministic Büchi automaton A for $\neg\varphi$,
2. Transform A to a non-deterministic Büchi automaton B for the negated Pnueli-Rosner closure of φ, i.e., the language of B is $\langle\,\mathsf{exp}\,\rangle\mathcal{L}(A) = \langle\,\mathsf{exp}\,\rangle(\neg\varphi)$,
3. Consider the structure of B as that of a *universal* co-Büchi automaton, which has language $\neg\mathcal{L}(B)$,
4. Synthesize an implementation f in the synchronous model which satisfies $\neg\mathcal{L}(B) = \neg\langle\,\mathsf{exp}\,\rangle\mathcal{L}(A) = \neg\langle\,\mathsf{exp}\,\rangle(\neg\varphi) = [\,\mathsf{exp}\,]\varphi = \mathsf{PR}(\varphi)$.

The new step is the second one, which constructs B from A; the others use standard constructions and tools. This construction is as follows.

- The states and alphabet of B are the states and alphabet of A.
- The transitions of B are determined by a saturation procedure. For every pair of states q, q', and letter (x, y), let $\Pi(q, (x, y), q')$ be the set of paths in A from q to q' where the sequence of letters on the path matches the expansion pattern $(\bot, y)^*(x, y)(\bot, y)^*$. The transition $(q, (x, y), q')$ is in B if, and only if, this set is non-empty,
- If some path in $\Pi(q, (x, y), q')$ passes through a green (accepting) state of A, the transition $(q, (x, y), q')$ in B is colored "green" and that path is assigned as the witness to the transition in B. On the other hand, if none of the paths in $\Pi(q, (x, y), q')$ pass through a green state, this transition is not colored in B, and one of the paths in the set is chosen as the witness for this transition,
- The automaton B inherits the accepting ("green") states of A and it may have, in addition, green transitions introduced as defined above,
- A sequence is accepted by B if there is a run of B on the sequence such that either there are infinitely many green states, or infinitely many green transitions on that run.

We establish that $\mathcal{L}(B) = \langle\,\mathsf{exp}\,\rangle\mathcal{L}(A)$ through the following two lemmas.

Lemma 1. $\langle\,\mathsf{exp}\,\rangle\mathcal{L}(A) \subseteq \mathcal{L}(B)$.

Proof. Let $\delta = (x_0, y_0)(x_1, y_1)\ldots$ be a sequence in $\langle\,\mathsf{exp}\,\rangle\mathcal{L}(A)$. By definition, there exists a sequence σ in $\mathcal{L}(A)$ such that $\delta\,\mathsf{exp}\,\sigma$. The expansion σ follows the pattern $[(\bot, y_0)^*(x_0, y_0)(\bot, y_0)^*][(\bot, y_1)^*(x_1, y_1)(\bot, y_1)^*]\ldots$, where $[\ldots]$ are used merely to indicate the boundaries of a block. An accepting run of A on σ has the form $q_0[(\bot, y_0)^*(x_0, y_0)(\bot, y_0)^*]q_1[(\bot, y_1)^*(x_1, y_1)(\bot, y_1)^*]q_2\ldots$, where the states on the run inside a block have been elided. By the definition of B,

the segment $q_0(\bot, y_0)^*(x_0, y_0)(\bot, y_0)^* q_1$ induces a transition from q_0 to q_1 in B on the letter (x_0, y_0). Similarly, the following segment induces a transition from q_1 to q_2 on letter (x_1, y_1), and so forth. These transitions together form a run $q_0(x_0, y_0)q_1(x_1, y_1)q_2 \ldots$ of B on δ.

If one of the $\{q_i\}$ is green and appears infinitely often on the run on σ, the induced run on δ is accepting. Otherwise, as the run on σ is accepting, some green state of A occurs in the interior of infinitely many segments of that run. The transitions of B induced by those segments must be green, so the corresponding run on δ has infinitely many green edges, and is accepting for B. □

Lemma 2. $\mathcal{L}(B) \subseteq \langle \mathsf{exp} \rangle \mathcal{L}(A)$.

Proof. Let δ be accepted by B. We show that there is σ such that $\delta \,\mathsf{exp}\, \sigma$ and σ is accepted by A. Let δ have the form $(x_0, y_0)(x_1, y_1) \ldots,$. Denote the accepting run of B on δ by $r = q_0(x_0, y_0)q_1(x_1, y_1) \ldots$. From the construction of B, the transition from q_0 to q_1 on (x_0, y_0) has an associated witness path through A from q_0 to q_1, which follows the expansion pattern $(\bot, y_0)^*(x_0, y_0)(\bot, y_0)^*$ on its edge labels. Stitching together the witness paths for each transition of r, we obtain both a sequence σ that is an expansion of δ and a run r' of A on σ.

As r is accepting for B, it must enter infinitely often either a green state or a green edge. If it enters a green state infinitely often, that state appears infinitely often on r'. If r enters a green edge infinitely often, the witness path for that edge contains a green state of A, say q; as this path is repeated infinitely often on σ, q appears infinitely often on r'. In either case, a green state of A appears infinitely often on r', which is therefore, an accepting run of A on σ. □

Automaton B can be placed in standard form by converting its green edges to green states as follows, forming a new automaton, $\hat{B}$. Form a green copy of the state space, i.e., for each state q, form a green variant, $G(q)$, which is marked as an accepting state. Set up transitions as follows. If (q, a, q') is an original non-green transition, then (q, a, q') and $(G(q), a, q')$ are new transitions. If (q, a, q') is an original green transition, then $(q, a, G(q'))$ and $(G(q), a, G(q'))$ are new transitions. This at most doubles the size of the automaton. It is straightforward to establish that $\mathcal{L}(B) = \mathcal{L}(\hat{B})$.

3.4 Symbolic Construction

The symbolic construction of $\hat{B}$ closely follows the definitions above. It is easily implemented with BDDs representing predicates on the input and output variables x and y. The crucial step is to use fixpoints to formulate the existence of paths in the set Π used in the definition of B. These definitions are similar to the fixpoint definition of the CTL modality EF. We use $A(q, (x, y), q')$ to denote the predicate on (x, y) describing the transition from q to q' in automaton A.

Fixed don't-care path. Let $\mathsf{EfixedY}(q, y, q')$ hold if there is a path of length 0 or more from q to q' in A where the value of y is fixed. This is the least fixpoint (in Z) of the following implications:

– $(q' = q) \Rightarrow Z(q, y, q')$, and
– $(\exists x, r : A(q, (x, y), r) \wedge Z(r, y, q')) \Rightarrow Z(q, y, q')$

The predicate $A^{\perp}(q, y, r) = (\exists x : A(q, (x, y), r))$ is pre-computed. Then, the least fixpoint is computed iteratively as follows.

$$\mathsf{EfixedY}^0(q, y, q') = (q = q')$$
$$\mathsf{EfixedY}^{i+1}(q, y, q') = \mathsf{EfixedY}^i(q, y, q') \vee (\exists r : A^{\perp}(q, y, r) \wedge \mathsf{EfixedY}^i(r, y, q'))$$

Let predicate $\mathsf{green}_A(r)$ be true for an accepting state r of A. The predicate $\mathsf{Efixedgreen}(q, y, q')$ holds if there is a fixed y-path from q to q' where one of the states on it is green:

$$\mathsf{Efixedgreen}(q, y, q') = (\exists r : \mathsf{EfixedY}(q, y, r) \wedge \mathsf{green}_A(r) \wedge \mathsf{EfixedY}(r, y, q'))$$

Paths and Green Paths. Let $\mathsf{Epath}(q, (x, y), q')$ hold if there is a path following the block pattern $(\perp, y)^*(x, y)(\perp, y)^*$ from q to q' in A. Then,

$$\mathsf{Epath}(q, (x, y), q') = (\exists r, r' : \mathsf{EfixedY}(q, y, r) \wedge A(r, (x, y), r') \wedge \mathsf{EfixedY}(r', y, q'))$$

Similarly, let $\mathsf{Egreenpath}(q, (x, y), q')$ hold if there is a path following the block pattern $(\perp, y)^*(x, y)(\perp, y)^*$ from q to q' in A, with an intermediate green state.

$$\begin{aligned}\mathsf{Egreenpath}(q, (x, y), q') = \\ &(\exists r, r' : \mathsf{Efixedgreen}(q, y, r) \wedge A(r, (x, y), r') \wedge \mathsf{EfixedY}(r', y, q')) \vee \\ &(\exists r, r' : \mathsf{EfixedY}(q, y, r) \wedge A(r, (x, y), r') \wedge \mathsf{Efixedgreen}(r', y, q'))\end{aligned}$$

State space of $\hat{B}$. The state space of $\hat{B}$ is formed by pairs (q, g), where q is a state of A and g is a Boolean indicating whether it is a new green state. The accepting condition $\mathsf{green}_{\hat{B}}(q, g)$ of $\hat{B}$ is given by $\mathsf{green}_A(q) \vee g$.

Initial states. The initial predicate $I_{\hat{B}}(q, g)$ is $I_A(q) \wedge \neg g$, where $I_A(q)$ is true for initial states of the input automata A.

Transition relation of $\hat{B}$. The transition relation $\hat{B}((q, g), (x, y), (q', g'))$ is

$$\hat{B}((q, g), (x, y), (q', g')) = \mathsf{Epath}(q, (x, y), q') \wedge (g' \equiv \mathsf{Egreenpath}(q, (x, y), q'))$$

4 Implementation and Experiments

The PR algorithm has been implemented in a framework called BAS (Bounded Asynchronous Synthesis). It uses the LTL-to-automaton converter LTL3BA [3,6], and follows the modular method, connecting to either of two solvers, BoSy [2,12] and Acacia+ [1,8] to solve the synchronous realizability of $\mathsf{PR}(\varphi)$. The PR construction is implemented in about 1200 lines of OCaml, using an external BDD library. (The core construction requires only about 400 lines of code.) For an LTL specification φ, the BAS workflow for asynchronous synthesis is as follows:

	Specification	Asyn. realizable?	PR constr.	Asyn. synthesis	
				BoSy	Acacia+
1	$\Box\,(x \equiv y)$	false	8	972	30
2	$\Diamond\Box\, x \equiv \Diamond\Box\, y$	false	9	Na	Na
3	$\Diamond\Box\, x \Rightarrow \Diamond\Box\, y$	true	8	899	Na
4	$\Diamond\Box\, y \Rightarrow \Diamond\Box\, x$	true	7	994	Na
5	$(\Diamond\Box\, x \vee \Diamond\Box\, \neg x) \Rightarrow \Diamond\Box\, x \equiv \Diamond\Box\, y$	true	13	1004	Na
6	$\Box\,(\neg x \Rightarrow (\neg x)\ \mathsf{U}\ (\neg y)) \Rightarrow \Diamond\Box\, x \equiv \Diamond\Box\, y$	true	10	Na	Na
7	$\Box\Diamond\,(x \wedge y) \Rightarrow (\Box\Diamond\, y \wedge \Box\Diamond\, \neg y)$	true	9	1053	30
8	$\Box\Diamond\,(x \vee y) \Rightarrow (\Box\Diamond\, y \wedge \Box\Diamond\, \neg y)$	true	9	995	40
9	$\Box\Diamond\,(x) \Rightarrow (\Box\Diamond\, y \wedge \Box\Diamond\, \neg y)$	true	8	934	30
10	$\Box\,(x \Rightarrow \Diamond\, y)$	true	8	960	30
11	$\Box\,(x \Rightarrow \Diamond\, y) \wedge \Box\,(\neg y\ \mathsf{U}\ x)$	false	10	1058	Na
	Variants of parameterized arbiter (results shown are for $n = 2; 4; 6$)				
12	$\bigwedge_{i\neq j} \Box\,(\neg g_i \vee \neg g_j) \quad \wedge$ $\bigwedge_{i=1}^{n} \Box\,(r_i \Rightarrow \Diamond\, g_i)$	true	11; 13; 75	854; 1146; 4965	Na; Na; Na
13	$\bigwedge_{i\neq j} \Box\,(\neg g_i \vee \neg g_j) \quad \wedge$ $\bigwedge_{i=1}^{n} \Box\,(r_i \Rightarrow \Diamond\, g_i) \wedge \bigwedge_{i=1}^{n} \Box\,(g_i \Rightarrow r_i)$	false	17; 3124; 2024K	1129; 362K; Na	Na; Na; Na

Table 1. BAS asynchronous synthesis runtime evaluation (times in milliseconds). We let BoSy run upto 2 hours, and Acacia+ upto 1000 iterations. "Na" denotes cases where the executions did not find a winning strategy within these boundaries.

1. Check whether φ is synchronously realizable; if not, return `UNREALIZABLE`,
2. Construct Büchi automata A for $\neg\varphi$, and $\hat{A}$ for φ,
3. Concurrently
   (a) Construct $\mathsf{PR}(\varphi)$ from A and check whether it is synchronously realizable; if so, return `REALIZABLE` and synthesize the implementation.
   (b) Construct $\mathsf{PR}(\neg\varphi)$ from $\hat{A}$ and check whether it is synchronously realizable for the environment; if so, return `UNREALIZABLE`.

   Upon termination of any, terminate the other execution as well.

The synchronous synthesis tools successively increase a bound until a limit (computed based on automaton structure) is reached. Thus, in theory, only the check in step 3(a) is needed. However, the checks in steps 1 and 3(b) may allow the tool to terminate early (before reaching the limit bound), if a winning strategy for the environment can be discovered.

To evaluate BAS we consider the list of examples presented in Table 1. The reported experiments were performed on a VM configured to have 8 CPU cores at 2.4GHz, 8GB RAM, running 64-bit Linux. The running times are reported in milliseconds. For each specification (presented in the second column) we report whether it is asynchronously realizable (third column), the time for the PR construction (our contribution), and the time for checking whether the specification is realizable using BoSy and Acacia+ solvers (resp., fifth and sixth columns).

The first set of examples (Specifications 1-11) list specifications discussed in this paper and in related works. As parameterized example we consider 2 variants of arbiter specifications. The arbiter has n inputs in which clients request permissions, and n outputs in which the clients are granted permissions. In both variants of the arbiter example, no two grants are allowed to be set simultaneously. The first arbiter example (Specification 12) requires that whenever an input request r_i is set, the corresponding output grant g_i must eventually be set. The second variant (Specification 13) also requires that a grant g_i is set only if request r_i is set as well. That is, in order for a client to be granted a permission, its corresponding request must be constantly set. Since the asynchronous case cannot observe the request in between read events, this variant of the arbiter is not realizable. The results are shown for $n = 2, 4, 6$. Note that the only comparable experimental evaluation is given in [18], where they report that asynchronous synthesis of the first arbiter example (Specification 12) takes over 8 hours.

The second specification φ is the one discussed in Section 2. It is surprisingly difficult to solve. Both φ and its negation are asynchronously unrealizable. Moreover, φ is synchronously realizable. Thus, the early detection tests (steps 1 and 3(b)) failed to discover a winning strategy for the environment; the bounded synthesis tools increase the considered bound monotonically without converging to an answer in a reasonable amount of time. This example highlights the need for better tests for unrealizability. The results in the following section provide simple QBF tests of unrealizability for subclasses of LTL.

5 Efficiently Solvable Subclasses of LTL

The high complexity of direct LTL (synchronous) synthesis has encouraged the search for general procedures that work well in practice, such as Safraless and bounded synthesis [24,35]. Another useful direction has been to identify fragments of LTL with efficient synthesis algorithms [5]. Among the most noteworthy is the GR(1) subclass, for which there is an efficient, symbolic synthesis procedure ([28]). We explore this direction for *asynchonous* synthesis. Surprisingly, we show that synthesis for certain fragments of LTL can be reduced to Boolean reasoning over properties in QBF. The results cover several types of GR(1) formulae, although the question of a reduction for all of GR(1) is open.

The QBF formulae that arise have the form $\exists y \forall x.p(x, y)$, where x and y are disjoint sets of variables, and p is a propositional formula over x, y. An assignment $y = b$ for which $\forall x.f(x, b)$ holds is called a *witness* to the formula. The first such reduction is for the property $\square \lozenge P$.

Theorem 4. *$\varphi = \square \lozenge P$ is asynchronously realizable iff $\exists y \forall x P$ is* True.

Proof. (ping) Let b be a witness to $\exists y \forall x.P$. The function that constantly outputs $y = b$ satisfies φ for any asynchronous schedule.

(pong) Let f be a candidate implementation function and suppose that $\forall y \exists x(\neg P)$ holds. Fix any schedule. For every value $y = b$ that function f outputs at a writing point, there exists an input value $x = a$ such that $\neg P(a, b)$ holds.

Thus, the environment, by issuing $x = a$ in the interval from the current writing point (with $y = b$) up to the next one, can ensure that $\neg P$ holds throughout the execution. Thus the specification $\varphi = \Box \Diamond P$ does not hold on this execution. $\Box$

The result in Theorem 4 applies to asynchronous synthesis, but does not apply to synchronous synthesis. For example, the property $\Box \Diamond (x \equiv y)$ is asynchronously unrealizable, as $\exists y \forall x (x \equiv y)$ is False. On the other hand, it is synchronously realizable with a Mealy machine that sets y to x at each point.

Theorem 4 extends easily to conjunction and disjunction of $\Box \Diamond$ properties.

Theorem 5. *Specification* $\varphi = \bigvee_{i=0}^{m} \Box \Diamond P_i$ *is asynchronously realizable iff* $\exists y \forall x . (\bigvee_{i=0}^{m} P_i)$ *holds. Additionally, specification* $\varphi = \bigwedge_{i=0}^{m} \Box \Diamond P_i$ *is asynchronously realizable iff for all* $i \in \{0, 1 \ldots m\}$, $\exists y \forall x . P_i$ *holds.*

Proof. The first claim follows directly from the identity $\bigvee_{i=0}^{m} \Box \Diamond P_i \equiv \Box \Diamond (\bigvee_{i=0}^{m} P_i)$ and Theorem 4.

For the second, for each i, let $y = b_i$ be an assignment such that $\forall x . P_i(x, b_i)$ holds. The function that generates sequence $b_0, b_1, \ldots b_m$, ad infinitum, is an asynchronous implementation of $\bigwedge_{i=0}^{m} \Box \Diamond P_i$. On the other hand, suppose that for some i, $\forall y \exists x \neg P_i$ holds, then following the construction from Theorem 4, one can define an execution where P_i is always False. $\Box$

Theorem 6. $\varphi = \Diamond \Box P$ *is asynchronously realizable iff* $\exists y \forall x . P$ *is* True.

The proof is similar to that for Theorem 4. Theorem 6 also extends to conjunctions and disjunctions of $\Diamond \Box$ properties, by arguments similar to those for Theorem 5. Namely, $\bigwedge_{i=0}^{m} \Diamond \Box P_i$ is asynchronously realizable iff $\exists y \forall x (\bigwedge_{i=0}^{m} P_i)$ is True, and, $\bigvee_{i=0}^{m} \Diamond \Box P_i$ is asynchronously realizable iff for some $i \in \{0, 1, \ldots m\}$, $\exists y \forall x . P_i$ is True. Theorems 4 - 6 apply to non-atomic reads and writes of multiple input and output variables. Proofs are in the full version of the paper.

We now consider a more general type of GR(1) formula. The *strict semantic* of GR(1) formula $\Box S_e \wedge \Box \Diamond P \Rightarrow \Box S_s \wedge \Box \Diamond Q$ is defined to be $\Box(\boxminus S_e \Rightarrow S_s) \wedge (\Box S_e \wedge \Box \Diamond P \Rightarrow \Box \Diamond Q)$ – i.e., S_s is required to hold so long as S_e has always held in the past; and if S_e holds always and P holds infinitely often, then Q holds infinitely often. This is the interpretation supported by GR(1) synchronous synthesis tools.

Theorem 7. *The strict semantics of GR(1) specification* $\Box S_e \wedge \Box \Diamond P \Rightarrow \Box S_s \wedge \Box \Diamond Q$ *is asynchronously realizable iff* $\exists y \forall x . (S_e \Rightarrow (S_s \wedge (P \Rightarrow Q)))$ *is* True.

Proof. (ping) If $y = b$ is a witness to $\exists y \forall x . (S_e \Rightarrow (S_s \wedge (P \Rightarrow Q)))$, let f be a function that always generates b. Suppose S_e holds up to point i, then as $y = b$, regardless of the x-value, S_s holds at point i. This shows that the first part of the specification holds. For the second, suppose that S_e holds always and P is true infinitely often. Then, by choice of $y = b$, $(P \Rightarrow Q)$ holds always, thus Q holds infinitely often as well.

(pong) To prove the other side of the implication, we proceed as in Theorem 4. Let f be a candidate implementation. Fix a schedule, and suppose that

$\forall y \exists x.(S_e \wedge (\neg S_s \vee \neg(P \Rightarrow Q)))$ holds. Then for every step of the execution and for every value $y = b$ that function f outputs at a writing point, there exists a value $x = a$ which the environment can choose from that writing point to the next such that $S_e(a, b)$ is true, and one of $S_s(a, b)$ or $(P \Rightarrow Q)(a, b)$ is false at every point in that interval.

On this execution, S_e holds throughout. If S_s is false at some point, this violates the first part of the specification. If not, then $(P \Rightarrow Q)$ must be false everywhere; i.e., at every point P is true but Q is false. Thus, S_e holds everywhere and P holds infinitely often but Q does not hold infinitely often, violating the second part of the specification. □

Theorem 7 applies to atomic reads and writes, showing that asynchronous synthesis of GR(1) specification can be reduced to Boolean reasoning over properties in QBF. For non-atomic reads and writes, safety in asynchronous systems is more nuanced, since there is a delay between the write points of the first and last outputs in each round. This is discussed in the full version of the paper. This result does not generalize easily to the full GR(1) format, where more than one $\square\lozenge$ property can appear on either side of the implication.

These results establish that the asynchronous synthesis problem for such specifications is easily solvable–more easily than in the synchronous setting, surprisingly avoiding entirely the need for automaton constructions and bounded synthesis. From another, equally valuable, point of view, the results show that such types of specifications may be of limited interest for automated synthesis, as solvable cases have very simple solutions.

6 Conclusions and Related Work

This work tackles the task of asynchronous synthesis from temporal specifications. The main results are a new symbolic automaton construction for general temporal properties, and the reduction of the synthesis question for several classes of specifications to QBF. These are mathematically interesting, being substantial simplifications of prior methods. Moreover, they make it feasible to implement an asynchronous synthesis tool following the modular process suggested by Pnueli and Rosner in 1989, by reducing asynchronous synthesis to a synchronous synthesis question. To the best of our knowledge, this is the first such tool. The prototype, which builds on tools for synchronous synthesis, is able to quickly synthesize asynchronous programs for several interesting properties. There are, undoubtedly, several challenges that remain, one of which is the quick detection of unrealizable specifications.

Our work builds upon several earlier results, which we discuss here. The synthesis question for temporal properties originates from a question posed by Church in the 1950s (see [37]). The problem of synthesizing a synchronous reactive system from a linear temporal specification was formulated and studied by Pnueli and Rosner [31], who gave a solution based on non-emptiness of tree automata. There has been much progress on the synchronous synthesis question

since. Key developments include the discovery of efficient symbolic (BDD-based) solutions for the GR(1) class [7,28], the invention of "Safraless" procedures [24], the application of these ideas for bounded synthesis [15,35], and their implementation in a number of tools, e.g. [8,10,11,13,20,34]. These have been applied in many settings (cf. [9,23,25,26,27]).

The problem of synthesizing asynchronous programs was also formulated and studied by Pnueli and Rosner [32] but has proved to be much more challenging, with only limited progress. The original Pnueli-Rosner constructions are complex and were not implemented. Work by Klein, Piterman and Pnueli, nearly 20 years later [22], shows tractability for some GR(1) specifications. However, the class of specifications that can be so handled is characterized by semantic constraints such as stuttering-closure and memoryless-ness, which are difficult to recognize.

Finkbeiner and Schewe [18,35] present an alternative method, based on bounded synthesis, that applies to all LTL properties: it encodes the existence of a deductive proof for a bounded program into SAT/SMT constraints. However, the encoding represents inputs and outputs explicitly and is, therefore, exponential in the number of input and output bits. The exponential blowup has practical consequences: an asynchronous arbiter specification requires over 8 hours to synthesize [18]; the same specification can be synthesized by our method in seconds. (Note, however, that the method in [18] is not specialized to asynchronous synthesis, and this difference may not be solely due to the explicit state representation, as the specification has only 4 bits.) Recent work gives an alternative encoding of synchronous bounded synthesis into QBF constraints, retaining input and output bits in symbolic form [12]. We believe that a similar encoding applies to asynchronous bounded synthesis as well, this is a topic for future work.

Pnueli and Rosner's model of interface communication is not the only choice. Other models for asynchrony could, for instance, be based on CCS/CSP-style rendezvous communication at the interface, or permit shared read-write variables with atomic lock/unlock actions. Petri net game models have also been suggested for distributed synthesis [16]. An orthogonal direction is to weaken the adversarial power of the environment through a probabilistic model which can be used to constrain unlikely, highly adversarial input patterns to have probability 0, thus turning the synthesis problem into one where programs satisfy their specifications with high probability. (The synthesis of multiple processes is known to be undecidable in most cases [17,33].)

In the broader context of fully automatic program synthesis, there are various approaches to the synthesis of single-threaded, terminating programs from formal pre- and post-condition specifications and from examples, using type information and other techniques to prune the search space. (We will not attempt to survey this large field, some examples are [14,19,36].) An intriguing question is to investigate how the techniques developed in these distinct lines of work can be fruitfully combined to aid the development of asynchronous, reactive software.

Acknowledgements. Kedar Namjoshi was supported, in part, by NSF grant CCF-1563393 from the National Science Foundation. We would like to thank Michael Emmi for many helpful discussions during the early stages of this work.

References

1. Acacia+. http://lit2.ulb.ac.be/acaciaplus//.
2. BoSy. https://www.react.uni-saarland.de/tools/bosy/.
3. LTL3BA. https://sourceforge.net/projects/ltl3ba/.
4. Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
5. Rajeev Alur and Salvatore La Torre. Deterministic generators and games for ltl fragments. *Transactions on Computational Logic*, 5(1):1–25, 2004.
6. Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In *TACAS*, pages 95–109, 2012.
7. Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.
8. Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In *Proc. of CAV*, volume 7358, pages 652–657. Springer, 2012.
9. Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *Transactions on Software Engineering and Methodology*, 22(1):9, 2013.
10. Rüdiger Ehlers. Symbolic bounded synthesis. In *Proc. of CAV*, volume 6174, pages 365–379. Springer, 2010.
11. Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In *Proc. of TACAS*, pages 272–275. Springer, 2011.
12. Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. Encodings of bounded synthesis. In *TACAS*, pages 354–370, 2017.
13. Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 325–332. Springer, 2017.
14. Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017.
15. Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Compositional algorithms for LTL synthesis. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2010.
16. Bernd Finkbeiner and Ernst-Rüdiger Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017.
17. Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 321–330. IEEE Computer Society, 2005.
18. Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.
19. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In Rastislav Bodík

and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815. ACM, 2016.

20. Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design, 6th International Conference, FM-CAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 117–124. IEEE Computer Society, 2006.
21. Uri Klein. *Topics in Formal Synthesis and Modeling.* PhD thesis, New York University, 2011.
22. Uri Klein, Nir Piterman, and Amir Pnueli. Effective synthesis of asynchronous systems from GR(1) specifications. In *International Conference on VMCAI*, pages 283–298. Springer, 2012.
23. Hadas Kress-Gazit and George J Pappas. Automatic synthesis of robot controllers for tasks with locative prepositions. In *International Conference on Robotics and Automation (ICRA)*, pages 3215–3220, 2010.
24. Orna Kupferman and Moshe Y Vardi. Safraless decision procedures. In *Proc. of FOCS*, pages 531–540. IEEE, 2005.
25. Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 58(7):1771–1785, 2013.
26. Shahar Maoz and Yaniv Sa'ar. Aspectltl: an aspect language for LTL specifications. In Paulo Borba and Shigeru Chiba, editors, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*, pages 19–30. ACM, 2011.
27. Shahar Maoz and Yaniv Sa'ar. Assume-guarantee scenarios: Semantics and synthesis. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 2012.
28. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. In *International Conference on VMCAI*, volume 3855, pages 364–380. Springer, 2006.
29. Amir Pnueli. The temporal logic of programs. In *Proc. of FOCS*, pages 46–57. IEEE, 1977.
30. Amir Pnueli and Uri Klein. Synthesis of programs from temporal property specifications. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*, pages 1–7. IEEE, 2009.
31. Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
32. Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. *Automata, Languages and Programming*, pages 652–671, 1989.
33. Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 746–757. IEEE Computer Society, 1990.
34. Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. JTLV: A framework for developing verification algorithms. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 171–174. Springer, 2010.

35. Sven Schewe and Bernd Finkbeiner. Bounded synthesis. *Proc. of ATVA*, pages 474–488, 2007.
36. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326. ACM, 2010.
37. Wolfgang Thomas. Facets of Synthesis: Revisiting Church's Problem. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5504 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009.