Assume-Guarantee Distributed Synthesis

Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Damien Zufferey

***Abstract*—Distributed reactive synthesis is the problem of algorithmically constructing controllers of distributed, communicating systems so that each closed loop system satisfies a given temporal specification. We present an algorithm, called *negotiation*, for sound (but necessarily incomplete) distributed reactive synthesis based on assume-guarantee decompositions. The negotiation algorithm iteratively constructs assumptions and guarantees for each system. In each iteration, each system attempts to fulfill its specification and its guarantee (from the previous round), under the current assumption on the other systems, by solving a reactive synthesis problem. If the specification is not realizable, the algorithm computes a sufficient assumption on the other systems that ensures it can realize the specification and guarantee. This additional assumption further constrains the behavior of other systems and they might require an additional assumption, leading to the next round in the negotiation. The process terminates when a *compatible* assumption-guarantee pair is found for each system, which is sufficient to also satisfy the specification of each system. We have built a tool called Agnes that implements this algorithm. Using Agnes, we empirically demonstrate the effectiveness of our proposed algorithm on two case studies.**

I. Introduction

We consider the problem of *distributed* synthesis of reactive controllers for communicating systems connected in feedback against local temporal specifications. We consider a setting where each system reads and writes variables which can be either local or shared. The communication occurs through shared variables. In particular, a system only has a partial view of the overall state: it can look at its own state and the values of shared variables written by other systems or by the environment (but not the states of other systems). Each system has a local specification given by a language over valuations of its own state variables. We require local controllers for each system—those that make their decision solely based on locally available information—such that each system satisfies its own specification.

Distributed reactive synthesis is a well-studied problem, going back to the seminal paper of Pnueli and Rosner [31]. Unfortunately, for most distributed architectures, including the one studied in this paper, the problem of checking if distributed controllers exist is undecidable [31]. Even when the problem is decidable, e.g., when systems are connected in a pipeline fashion, the complexity of reactive synthesis is non-elementary. Thus, we must look for sound, but possibly incomplete, heuristics.

There are very few automated distributed synthesis tools available today. Among notable exceptions is *bounded* synthesis [19], in which the existence of distributed controllers up to a certain size is reduced to a constraint satisfaction problem in (quantified) Boolean logic; from the validity of the constraint system, one can read off distributed controllers if they exist.

In this paper, we consider an alternate, *modular* approach, based on *assume-guarantee decompositions*. *Assume-guarantee* proof systems provide a compositional approach to synthesis [4], [9], [10], [12], [14]. In this setting, each system makes an assumption on the temporal behavior of the other systems whose output it observes, and in exchange, provides a guarantee on its own behaviors. A strategy for the controller is then synthesized to enforce each local specification along with the guarantee under the hypothesis that the other systems respect the assumptions. If the behaviors under the synthesized strategy satisfies the guarantee and, moreover, the guarantee of each system implies the assumption used by the other systems, one can prove that the entire closed-loop system, using the synthesized strategies, satisfies all local specifications.

Usually, assume-guarantee proof systems either require user-provided assumptions and guarantees or assume a strategy profile has already been constructed and that systems can synchronize on this joint strategy profile. We take a different route. We algorithmically synthesize assume-guarantee contracts for each system such that the composition of these contracts lead to the fulfillment of the given specification. In this paper, we only consider *safe* assume-guarantee contracts, and leave the study of more general ω-regular contracts as part of the future work.

The simplest assumptions are *true*, when a system makes no assumptions on the other systems. This reduces to synthesizing strategies for each system assuming the other systems behave arbitrarily. While sound, this procedure is not so useful in practice because often, the systems cannot accomplish their objectives by themselves and require some cooperation from the other systems. It is also not sufficient to use the local specification for each system as their guarantees. For example, if one of the systems has a trivial specification *true*, it can satisfy its local specification by playing arbitrarily; some of these plays may prevent the other systems from meeting their specification. Thus, our algorithm has to explore the space of assumptions and guarantees to find mutually beneficial strategies that enable all systems to satisfy their specifications.

We develop an iterative algorithm that refines assumptions and guarantees. The key to our algorithm is the notion of *environment assumptions*, minimal restrictions on the behavior of the environment for a system to ensure its own specification as well as its previously promised guarantees.

All the authors are with Max Planck Institute for Software Systems, Germany. Email: {rupak, kmallik, akschmuck, zufferey}@mpi-sws.org.

This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue.

We use environment assumptions originally characterized by Chatterjee et al. [15] in the context of centralized reactive synthesis. In each iteration, we construct minimal environment assumptions and use the assumptions found by one system as additional constraints (guarantees) on the behavior of the others. We iteratively refine assumptions and guarantees until convergence.

Our algorithm is sound: if we ever find *compatible* assumptions and guarantees, we can derive distributed controllers that solve the synthesis problem. Since the problem is undecidable, we cannot ensure termination, although we empirically demonstrate that the algorithm does terminate for several distributed synthesis problems.

In each iteration, we split the state space of each system into three regions: the *realizable* region, where it can achieve its specification without any cooperation from the environment, the *surely unrealizable* region, where it is never possible to fulfill the objective even when the environment cooperates, and the *maybe-realizable* region, where there is a winning strategy provided the other system agrees to co-operate. We solve a reactive synthesis problem on the maybe-realizable region to find restrictions on the other systems (e.g., using standard fixed point algorithms [17], [25]) and then compute environment assumptions (using the algorithm from [15]).

We have built the tool called Agnes, that implements our algorithm together with several optimizations to reduce the search space. Internally, we represent assumptions as automata. We heuristically approximate the automata with smaller ones, for different metrics. For example, we have found the so-called l-complete abstractions of languages [27], that preserve suffixes of length up to l, particularly effective in our experiments. In contrast to bounded synthesis [19], we do not need to compute the global state space of all systems and controller sizes. Instead, we show empirically that for several benchmarks, the assume-guarantee iteration converges quickly and finds compatible assume-guarantee contracts.

Related Work. Assume-guarantee proofs are a well-known approach to modular reasoning about systems [1], [2], [13], and there are many algorithms that automate the search for assumptions and guarantees in the context of verification. Assume-guarantee techniques in synthesis have been studied before in the embedded context (see the survey [7]), but only few papers consider the iterative and automatic construction of assume-guarantee pairs in the distributed synthesis context. In fact, most synthesis tools do not handle distributed synthesis.

Our algorithm uses the characterization by Chatterjee et al. [15] of minimal environment assumptions in case of an unrealizable centralized reactive synthesis problem. Unlike their paper, we use these assumptions iteratively in a distributed context. Assumptions are succinct representations of allowed behaviors of the other system.

Close to our work is the distributed synthesis algorithm of Alur et al. [5]. Their algorithm finds a solution to the distributed synthesis problem by iteratively solving local synthesis problems under the assumption that the other components cooperate. Afterwards, they build a product of the so obtained local strategies to resolve any conflicts that may arise. In contrast, our method is an alternate approach, where the systems do not need to publicly share their own strategies; instead they share the (un-)desirable properties of the other systems from their own perspectives. One advantage of doing this is that our approach is completely modular: a component can freely change its control strategy as long as its contracts are locally fulfilled.

There are also distributed synthesis algorithms which use assume-guarantee pairs expressed using GR(1) specifications [3], [4]. However, in their setting, information flows only one way between the systems and one of the systems has to realize its specification without any assumptions. Then the other can assume certain behaviors from the first. In contrast, we assume feedback composition and a circular proof rule. In addition, we work with maximal permissive environment assumptions, using the algorithms and characterization from [15], whereas the assumptions are syntactic patterns in [3]. Since we assume feedback composition, our algorithm, in contrast to theirs, is an iterative fixed point computation.

An important question in synthesis under assumptions is whether a system can "cheat" and win a game by invalidating an environment assumption, and many recent papers have proposed notions of compositional synthesis that prevents such cheating [11], [22], [24]. This is not a problem in our context, as our assumptions are over the external alphabet and a system cannot willfully prevent any environment behavior. Often, coordinated behavior is enforced through game-theoretic means that go beyond the classical setting of reactive synthesis [8], [12], [16], [18], [20], [23]; instead, we work in the classical Pnueli-Rosner setting.

II. Example: Distributed Shared Bus

We start with a simple example to describe our algorithm. Consider a distributed architecture with two synchronous systems C_0 and C_1, where each system attempts to transmit a single data packet through a shared bus within a certain deadline. Assume that sending each data packet takes one time unit for the bus. There is no handshaking involved in the transmission process: whenever a system wants to send the packet, it simply needs to write it at the sending end of the bus. However, if both systems write their data packet exactly at the same instant, the bus turns down the send request from both of the systems to avoid data corruption. When a send request is turned down by the bus, the systems can attempt to resend the failed data packet in the future.

Figure 1 shows the structure of one system in a guarded command language. The other system is similar. Each system has *state variables* (separate copies of s and t) that it reads and writes, *external variables* from the environment that it can read (env), and *output variables* that it writes and that provide its visible state to the environment (out). The external variables provide the visible state of the rest of the system—a system does not control their values. Additionally, the system has a number of input actions ($wait$ and wr) it can use to determine how its state is updated.

A state maps the state variables to values. Initially, the state is $(idle, 4)$: the system has 4 steps to send the packet. The transitions map the current state, current values of external

state var $s \in \{idle, writing, done\}$,
$t \in \{1, 2, 3, 4\}$
external var $env \in \{idle, busy\}$
output var $out \in \{idle, busy\}$
input action $U = \{wait, wr\}$

init $s = idle, t = 4$
transition
$[\!] \; s = idle \xrightarrow{wait} s' = idle \wedge t' = t - 1$
$[\!] \; s = idle \xrightarrow{wr} s' = writing \wedge t' = t - 1$
$[\!] \; s = writing \wedge t \geq 2 \wedge env = idle \xrightarrow{wr} s' = done$
$[\!] \; s = writing \wedge t \geq 2 \wedge env = busy \xrightarrow{wr}$
$s' = writing \wedge t' = t - 1$
output $out = busy$ if $s = writing$ and $idle$ otherwise

Fig. 1: The packet sender system. Our example has two such systems running synchronously in parallel.

variables, and current choice of control inputs to new values of the state variables. For example, when the state is $idle$, picking the $wait$ input action keeps the state idle but decreases t; picking the wr input action changes the state to write and also decreases t. The output variable is a function of the state; it is visible to other systems.

Intuitively, the system moves from $idle$ to $writing$ and then to $done$, once it successfully sends the packet. However, if the bus is busy, it may fail to send the packet by the deadline. Each system wants to eventually successfully write its data packet. In terms of state variables, and using linear temporal logic (LTL) notation, the specification is $\Diamond(s = done)$.

We are looking for a *distributed* solution to the problem: each system must run a local controller that only sees the state of the system and the history of external inputs that it receives. Thus, we cannot simply take the product of the individual state spaces and run a reactive synthesis algorithm for the conjunction of the local specifications.

Worst-Case Environment. Suppose we try to find a controller for each system, independently of the other. Unfortunately, we realize that there is no local controller without any assumptions on the behavior of the other system: in the worst case, the other system could be writing in all cycles, and our system will never be able to send its packet.

Assume the Specification of the Environment. Clearly, a "worst-case" behavior is too pessimistic. At least, the other system must satisfy its own specification. What if we assumed that the behavior of the other system is constrained by its own specification? Unfortunately, this is still not sufficient in our example: if we only know that the other system *eventually* does not write to the bus, we could still try to write in the same cycle. Moreover, both systems could end up waiting for each other.

Assumptions, Guarantees, and Negotiations. An intuitive solution to the problem is that one system promises to write only in even cycles and the other only in odd cycles. Then, the first system can make progress towards its write: it waits until the next even cycle and writes.

We have performed an *assume-guarantee decomposition* of the problem. Each system makes an assumption on the behavior of the other system, and under the assumption, it implements a controller that satisfies its specification and possibly a further constraint on its behavior—its guarantee to the other systems. If the guarantees of one system are contained in the assumptions made by the other, then we can show that there is a distributed controller implementation.

Our contribution is to show how we can automatically come up with such assume-guarantee pairs. In real life, whenever two entities, people or organizations, with their own set of interests need to make an agreement for a peaceful co-existence, a *negotiation* process is called for. Accordingly, we call our algorithm to iteratively compute assume-guarantee pairs a *negotiation*. We will use the above motivating example to give an informal description of our solution method.

The negotiation is an iterative procedure. Initially, we make no assumptions about the other system and check if perhaps each system can satisfy its specification no matter how the other one behaves; if so, we are done. On the other hand, if a system cannot satisfy its specification even while assuming full cooperation of the others, we can stop—certainly we shall not find any implementation in this case.

Otherwise, we proceed by finding an *environment assumption*: a restriction on the behavior of the other system that enables our system to satisfy its specification.

Let us look at the negotiation from the perspective of system C_0. For example, assuming $t = 4$, C_0 would find an assumption $A_0 = (busy_1 + idle_1) \cdot (busy_1 + idle_1) \cdot idle_1^\omega$, which states that system C_1 does not transmit *after* the second cycle. Under this assumption, C_0 can satisfy its specification: simply send the packet after the second cycle.

Next, we check if system C_1 can indeed satisfy its own specification while additionally guaranteeing the assumption A_0 of system C_0. We check this by defining the guarantee G_1 as the projection of A_0 to the output variables of C_1 and seeing if C_1 has a winning strategy in the game $\Diamond done \wedge G_1$. Unfortunately, since system C_1 makes no assumptions (its current assumption is "true"), it cannot fulfill this specification. We find a tighter environment assumption that C_0 must ensure in order for C_1 to win; it is the language $A_1 = (busy_0 + idle_0) \cdot idle_0 \cdot (busy_0 + idle_0)^\omega$, which states that system C_0 does not transmit *in* the second cycle.

In general, given a system's current assumption A_i and guarantee G_i, we check if it has a strategy to fulfill its specification $\Phi_i = \Diamond done$ under the contract (A_i, G_i). If not, we find a new assumption and update the other system's guarantee, which starts a new round of negotiation.

We show this process is sound: if both systems can win the above game, then the current assumptions and guarantees form an assume-guarantee decomposition, and we can "read off" a distributed controller implementation. In our example, the negotiation terminates in the second round and outputs these final assumptions and guarantees:

$$A_0 = (busy_1 + idle_1) \cdot (busy_1 + idle_1) \cdot idle_1^\omega \quad (1)$$
$$G_0 = (busy_0 + idle_0) \cdot idle_0 \cdot (busy_0 + idle_0)^\omega \quad (2)$$
$$A_1 = (busy_0 + idle_0) \cdot idle_0 \cdot (busy_0 + idle_0)^\omega \quad (3)$$
$$G_1 = (busy_1 + idle_1) \cdot (busy_1 + idle_1) \cdot idle_1^\omega \quad (4)$$

III. Assume-Guarantee Decompositions

A. Preliminary Definitions

For an alphabet Σ, we write Σ^* and Σ^ω to denote the set of finite and infinite words over Σ, and Σ^∞ the set $\Sigma^* \cup \Sigma^\omega$. We write ϵ for the empty word and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ for the set of non-empty finite words.

We define the prefix relation on words as $u \leq w$ if there exists v such that $uv = w$. Note that w can be an ω-word. We extend the notion to languages: the prefix of a language L, written $\mathrm{pref}(L)$, is the set $\{w \in \Sigma^* \mid \exists u \in L. w \leq u\}$. A language L is *prefix-closed* if, whenever $w \in L$ and $u < w$, then $u \in L$. Given a $(*)$-language $L \subseteq \Sigma^*$, we define the *limit* $\lim(L)$ of L as the ω-language $\{u \in \Sigma^\omega \mid \exists$ infinitely many n s.t.$u_{[1,n]} \in L\}$. Thus, an infinite word belongs to the limit of a $*$-language L iff infinitely many of its prefixes belong to L. If L is prefix-closed, this implies that an infinite word belongs to $\lim(L)$ iff all its finite prefixes belongs to L. An (ω-)language L is a *safety* language if $L = \lim(\mathrm{pref}(L))$ and a *liveness* language if $\mathrm{pref}(L) = \Sigma^*$.

We shall consider systems defined by *variables* ranging over a fixed finite domain. For a set of variables X, we write $\mathbf{X}$ (the same symbol, in blue bold face) for a valuation that maps each variable in X to a value in its domain. We need the following notation. Suppose X and Y are two disjoint sets of variables. We write $\mathbf{X} \times \mathbf{Y}$ for the joint valuation that maps each variable in $X \cup Y$ to a value in that variable's domain. For any given $x \in \mathbf{X} \times \mathbf{Y}$, we write $x[X]$ and $x[Y]$ for the restriction of x to the domain X and Y, respectively. Conversely, if $x \in \mathbf{X}$ and $y \in \mathbf{Y}$, we write $x \oplus y \in \mathbf{X} \times \mathbf{Y}$ for the valuation that maps variables in X and Y according to x and y respectively.

Let X be a set of variables. For a word $x = x^0 x^1 \ldots \in \mathbf{X}^\infty$, and a subset $Y \subseteq X$, we write the projection $\mathbf{proj}_Y(x)$ for the word $x^0[Y]x^1[Y]\ldots$ that restricts each x^i, $i \geq 0$, to the domain Y. For a relation $E \subseteq X \times Y$, we write $\mathrm{dom}\, E$ to denote the domain $\{x \in X \mid \exists y \in Y \,.\, (x, y) \in E\}$ of E.

B. Systems

A system $C = (X, x_{in}, U, W, \delta, Y, h)$ consists of a finite set X of *state variables*, an initial state $x_{in} \in \mathbf{X}$, a finite set of *input actions* U, a finite set of *external variables* W, a total transition function $\delta : \mathbf{X} \times \mathbf{W} \times U \to \mathbf{X}$, a finite set of *output variables* Y, and an *output labeling function* $h : \mathbf{X} \to \mathbf{Y}$. We assume the sets of variables are pairwise disjoint.

The *run* of a system C starting from a state x^0 is a sequence $\rho \equiv x^0 \xrightarrow{w^0,u^0} x^1 \xrightarrow{w^1,u^1} x^2 \xrightarrow{w^2,u^2} \ldots$, where $x^i \in \mathbf{X}$, $w^i \in \mathbf{W}$, $u^i \in U$ for each $i \geq 0$, and moreover we have $\delta(x^i, w^i, u^i) = x^{i+1}$ for each $i \geq 0$. Unless otherwise mentioned, we will always assume that a run starts at the initial state, i.e., $x^0 = x_{in}$. The output of the run is the sequence $h(x^0)h(x^1)\ldots$, which maps states to their output labels. Intuitively, first, the set of state variables are set to the initial value x^0. Then, in each step, the external variables in W are set to arbitrary values, and an action from U is picked.

The transition relation determines the new values of the state variables, and the output variables get their values from the state variables using the output labeling function. For the run ρ, we write $\mathbf{proj}_X(\rho)$ for the sequence $x^0 x^1 \ldots \in \mathbf{X}^\omega$, $\mathbf{proj}_W(\rho)$ for $w^0 w^1 \ldots \in \mathbf{W}^\omega$, $\mathbf{proj}_Y(\rho)$ for $h(x^0)h(x^1)\ldots \in \mathbf{Y}^\omega$, and for any given subset $V \subseteq W$, we write $\mathbf{proj}_V(\rho)$ for $w^0[V]w^1[V]\ldots \in \mathbf{V}^\omega$.

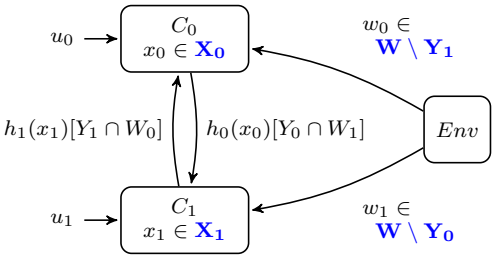

Fig. 2: A composition $C_0 \parallel C_1$

Let C_0 and C_1 be two systems; assume X_0, X_1, Y_0, and Y_1 are all disjoint. We define their parallel composition $C_0 \parallel C_1$ (see Fig. 2) as the system $(X, x_{in}, U_0 \times U_1, W, \delta, Y_0 \cup Y_1, h)$, where $X = X_0 \uplus X_1$, $x_{in} = x_{in0} \oplus x_{in1}$, $W = W_0 \cup W_1 \setminus (Y_0 \cup Y_1)$, $\delta : \mathbf{X} \times \mathbf{W} \times U_0 \times U_1 \to \mathbf{X}$, and $h : \mathbf{X} \to \mathbf{Y_0} \times \mathbf{Y_1}$, such that $h(x) = h_0(x[X_0]) \oplus h_1(x[X_1])$ and $\delta(x, w, (u_0, u_1)) = x'$ iff $\delta_0(x[X_0], (w \oplus h_1(x[X_1]))[W_0], u_0) = x'[X_0]$ and $\delta_1(x[X_1], (w \oplus h_0(x[X_0]))[W_1], u_1) = x'[X_1]$.

A run ρ of $C_0 \parallel C_1$ is a sequence

$$x^0 \xrightarrow{w^0,(u_0^0,u_1^0)} x^1 \to \ldots$$

where for each $i \geq 0$, we have $x^i \in \mathbf{X}$, $w^i = \mathbf{W}$, $u_0^i \in U_0$, $u_1^i \in U_1$, such that $x^0 = x_{in}$ and for each $i \geq 0$, we have $\delta(x^i, w^i, (u_0^i, u_1^i)) = x^{i+1}$.

Intuitively, the two systems C_0 and C_1 run synchronously in parallel. The state of the system is a valuation to the state variables of each system. In each step, an environment sets the values of variables in W. Each system sees the valuation to its external variables, which consist of the variables set by the environment as well as the output variables of the other system, picks an input action, and updates its state based on its transition function.

C. Distributed Realizability

Specifications and Realizability. A *specification* for a system is a language $\Phi \subseteq (\mathbf{X} \times \mathbf{W})^\omega$ that describes the correct runs; a run $x^0 \xrightarrow{w^0,u^0} x^1 \to \ldots$ *satisfies* a specification Φ if $(x^0, w^0)(x^1, w^1)\ldots \in \Phi$. A *local* specification is a language $\Phi \subseteq \mathbf{X}^\omega$. A run satisfies a local specification Φ if it satisfies the specification $\{(x^0, w^0)(x^1, w^1)\ldots \in (\mathbf{X} \times \mathbf{W})^\omega \mid x^0 x^1 \ldots \in \Phi\}$.

A *strategy* for a system is a function $\pi : (\mathbf{X} \times \mathbf{W})^+ \to U$. Likewise, an environment strategy is a function $\pi' : (\mathbf{X} \times \mathbf{W})^* \times \mathbf{X} \to \mathbf{W}$. A run $x^0 \xrightarrow{w^0,u^0} x^1 \to \ldots$ is compliant with π and/or π' if for each $i \geq 0$, we have $u^i = \pi(x^0, w^0, \ldots, x^i, w^i)$ and/or $w^i = \pi'(x^0, w^0, \ldots, x^{i-1}, w^{i-1}, x^i)$. We denote by $\rho(\pi, \pi')$ the unique run compliant with π and π'. Unless stated otherwise, a compliant run must start from $x^0 = x_{in}$.

A system C *can realize* a specification Φ (or Φ is realizable by C) if there is a system strategy π, called the *realization strategy for* Φ, such that for all runs ρ compliant with π holds

that $\mathbf{proj}_{X\times W}(\rho) \in \Phi$. Intuitively, we model the realizability of a specification as a two-player game between the system and the environment; the system can realize a specification if it has a realization strategy such that no matter how the environment plays, the resulting run belongs to the language of the specification.

We sometime use Linear Temporal Logic [6] notation to express local specifications. For example, given a set $B \subseteq \mathbf{X}$, we write $\Box B$ (safety: "always B"), $\Diamond B$ (reachability: "eventually B"), and $\Box\Diamond B$ (Büchi "eventually always B") to denote respectively the sets $\{x^0x^1\ldots \mid \forall i \geq 0\ .\ x^i \in B\} \subseteq \mathbf{X}^\omega$, $\{x^0x^1\ldots \mid \exists i \geq 0\ .\ x^i \in B\} \subseteq \mathbf{X}^\omega$, and $\{x^0x^1\ldots \mid \forall i \geq 0\ .\ \exists j \geq i\ .\ x^j \in B\} \subseteq \mathbf{X}^\omega$.

Distributed Realizability. Now consider a composition $C_0 \parallel C_1$. Suppose $\Phi_0 \subseteq \mathbf{X_0}^\omega$ and $\Phi_1 \subseteq \mathbf{X_1}^\omega$ are local specifications defined on the state variables of each system. One can define realization for the composition $C_0 \parallel C_1$ by considering a game between the composed system and the environment. However, such a centralized realization strategy may require coordination, e.g., to know the states of the two systems at some point. Instead, we want the realization strategies to be *distributed*: each component C_0 and C_1 should be able to pick their inputs based solely on the local history of valuations to state variables and their own environment inputs.

We model the distributed synthesis problem as a game (of incomplete information) between three players: system C_0, system C_1, and the environment. The game starts from the initial state x_{in} of $C_0 \parallel C_1$. In each step of the game, first, the environment picks values for the external variables in W, and then the systems C_0 and C_1 independently and simultaneously pick actions and the game proceeds to the next state.

We require that the strategies of system C_0 and system C_1 only depend on the history visible to them. Thus, we define a strategy of system C_i, $i \in \{0,1\}$, to be a function of the form $(\mathbf{X_i} \times \mathbf{W_i})^+ \to U_i$. Fixing strategies π_0, π_1, and π' of systems C_0, C_1, and the environment, respectively, yields a unique run $\rho(\pi_0, \pi_1, \pi')$ of the system $C_0 \parallel C_1$.

The *distributed synthesis* problem $(C_0, \Phi_0, C_1, \Phi_1)$ for the composition $C_0 \parallel C_1$ with local specifications $\Phi_0 \subseteq \mathbf{X_0}^\omega$ and $\Phi_1 \subseteq \mathbf{X_1}^\omega$ asks if there exist strategies $\pi_0 : (\mathbf{X_0} \times \mathbf{W_0})^+ \to U_0$ and $\pi_1 : (\mathbf{X_1} \times \mathbf{W_1})^+ \to U_1$ such that for all strategies $\pi' : (\mathbf{X} \times \mathbf{W})^* \times \mathbf{X} \to \mathbf{W}$, we have that $\mathbf{proj}_{X_0}(\rho(\pi_0, \pi_1, \pi')) \in \Phi_0$ and $\mathbf{proj}_{X_1}(\rho(\pi_0, \pi_1, \pi')) \in \Phi_1$. In that case, we say $C_0 \parallel C_1$ can *realize* the distributed synthesis problem.

Clearly, if C_0 and C_1 can each realize the local specifications Φ_0 and Φ_1 respectively, then $C_0 \parallel C_1$ can also realize the distributed synthesis problem. This is because the strategies do not make any assumptions on the behavior of the other system. However, it is possible that C_0 and C_1 do not each realize their specifications but they realize the distributed synthesis problem; for example, one system can use an assumption about the behavior of the other.

Unfortunately, the distributed synthesis problem is undecidable in general [31]. We summarize the discussion in the following proposition.

Proposition III.1. *(1) If for $i \in \{0,1\}$, the system C_i realizes Φ_i then the composition $C_0 \parallel C_1$ realizes the distributed synthesis problem $(C_0, \Phi_0, C_1, \Phi_1)$. (2) [31] The distributed synthesis problem is undecidable.*

We introduce some notation. For $i \in \{0,1\}$, we define the *realizable region*, denoted as $\langle\langle C_i \rangle\rangle \Phi_i$, as the largest subset of $\mathbf{X_i}$ such that for all states $x \in \langle\langle C_i \rangle\rangle \Phi_i$ there exists a run ρ compliant with a realization strategy π of Φ_i that visits x. Now consider a composition $C_0 \parallel C_1$. We say system C_i *can maybe-realize* Φ_i (or Φ_i is *maybe-realizable* by C_i) if there is a pair of (possibly co-ordinated) strategies π_0, π_1, called the *joint realization strategy*, such that for all strategies π' it holds that $\mathbf{proj}_{X_i}(\rho(\pi_0, \pi_1, \pi')) \in \Phi_i$. We define the *maybe-realizable region*, denoted as $\langle\langle C_0, C_1 \rangle\rangle \Phi_i$, as the largest subset of $\mathbf{X_i}$ such that for all states in $x \in \langle\langle C_i \rangle\rangle \Phi_i$ there exists a run ρ compliant with a *joint realization strategy* π_0, π_1 of Φ_i that visits x. We also define the *surely unrealizable region* as the complement of the maybe-realizable region.

D. Assume-Guarantee Contracts

Given a system C, an *assume-guarantee contract*—a *contract* in short—is a pair (A, G) of *safety* languages called the *assumption* $A \subseteq \mathbf{V}^\omega$ for some $V \subseteq W$, and the *guarantee* $G \subseteq \mathbf{X}^\omega$.

Definition III.2. *Let C be a system, and (A, G) be a contract. Then C can realize (A, G) if and only if there exists a system strategy π, such that for all $k > 0$ and for all finite runs $r \equiv x^0 \xrightarrow{w^0, u^0} x^1 \to \ldots x^k$ compliant with π,* either *of the following hold:*
(a) $\mathbf{proj}_X(r) \in \mathrm{pref}(G)$,
(b) there exists $0 \leq l < k$ such that $\mathbf{proj}_V(r)|_{[0,l]} \notin \mathrm{pref}(A)$. That is, a violation of G is preceded by a violation of A. The respective strategy π is called a realization strategy for (A, G).

For a contract (A, G) and specification Φ, we say C *can realize* Φ *under* contract (A, G), written C can realize the specification $\langle A \rhd \Phi \rhd G \rangle$, if there exists a strategy of C that is both a realization strategy for the contract (A, G) and a realization strategy for the specification $(A \Rightarrow \Phi)$. The maybe-realizability and sure unrealizability of a contract (A, G) and the specification $\langle A \rhd \Phi \rhd G \rangle$ are defined analogously.

Definition III.3. *Consider a system composition $C_0 \parallel C_1$. Let (A_0, G_0) and (A_1, G_1) be a pair of contracts for respectively C_0 and C_1 such that $\emptyset \subsetneq A_i \subseteq \mathbf{Y}^\omega_{1-i}$ and $\emptyset \subsetneq G_i \subseteq \mathbf{X}^\omega_i$. Then the contracts (A_0, G_0) and (A_1, G_1) are* compatible *if the following conditions are met for both $i \in \{0,1\}$:*
(a) $G_i \subseteq h_i^{-1}(A_{1-i})$, and (b) C_i realizes (A_i, G_i).

The composition of compatible contracts satisfies the following claim, motivated by [2], [13], [26], [28], [29], [32]:

Theorem III.4. ***[Assume Guarantee Decomposition]*** *Let $(C_0, \Phi_0, C_1, \Phi_1)$ be the input to a distributed synthesis problem. Let (A_0, G_0) and (A_1, G_1) be a pair of compatible contracts of C_0 and C_1 respectively. If C_0 can realize Φ_0 under (A_0, G_0) and C_1 can realize Φ_1 under (A_1, G_1), then $C_0 \parallel C_1$ can realize the distributed synthesis problem.*

Proof. We use the following notation. For a given run $\rho \equiv x^0 \xrightarrow{w^0, u^0} x^1 \to \ldots x^k \xrightarrow{w^k, u^k} x^{k+1} \to \ldots$ and for a

given $k \geq 0$, we write $r^k(\rho)$ for the prefix of the run $x^0 \xrightarrow{w^0,u^0} x^1 \xrightarrow{w^1,u^1} \ldots \xrightarrow{w^{k-1},u^{k-1}} x_k$ of length k.

▶ First, observe that a compatible contract implies that there exist two strategies π_0 and π_1 which fulfill the conditions in Def. III.2 for the individual systems C_0 and C_1. Let, for some strategy of the external environment, $\rho \equiv x^0 \xrightarrow{w^0,(u_0^0,u_1^0)} x^1 \rightarrow \ldots$ be a run of $C_0 \parallel C_1$ that is compliant with both π_0 and π_1. First, for both $i \in \{0,1\}$, we prove by induction that for every k, $\mathbf{proj}_{X_i}(r^k(\rho)) \in \mathrm{pref}(G_i)$; then because G_i is a safety language, it will be established that $\mathbf{proj}_{X_i}(\rho) \in G_i$.

▷ The base case: For $k = 0$, $\mathbf{proj}_{X_i}(r^k(\rho)) = x_{ini}$, and we see that Cond. (a) in Def. III.2 must hold. (Cond. (b) can not be true since l cannot be negative.)

▷ Induction step: Fix a $k \geq 0$ s.t. $\mathbf{proj}_{X_i}(r^k(\rho)) \in \mathrm{pref}(G_i)$ for both $i \in \{0,1\}$. That is, Cond. (a) in Def. III.2 holds. We show that the same is true for $k+1$. We obtain the following chain of implications: From the assumption we have $\mathbf{proj}_{X_i}(r^k(\rho)) \in \mathrm{pref}(G_i)$. With this, it follows from Def. III.3 (a) that $\mathbf{proj}_{Y_i}(r^k(\rho)) \in \mathrm{pref}(A_{1-i})$. This implies that for all $0 \leq l < k+1$, $\mathbf{proj}_{Y_i}(r^l(\rho)) \in \mathrm{pref}(A_{1-i})$. This in turn implies that Cond. (b) in Def. III.2 does not hold for $r^{k+1}(\rho)$ for both $i \in \{0,1\}$. Therefore, Cond. (a) must hold, which proves the induction step. ▶ With this we get $\mathbf{proj}_{X_i}(r^k(\rho)) \in \mathrm{pref}(G_i)$ for any $k \in N$. As G_i is a safety language, this implies $\mathbf{proj}_{X_i}(\rho) \in G_i$. Then it follows from Def. III.3 (a) that $\mathbf{proj}_{Y_i}(\rho) \in A_{1-i}$.

▶ Both systems C_i can additionally realize their specification Φ_i under the given contract if there exist strategies π_0 and π_1 which renders both contracts compatible (implying $\mathbf{proj}_{Y_i}(\rho) \in A_{1-i}$ over all its compliant traces from above) and additionally ensuring that Φ_i holds on all traces on which A_i holds. As the latter is always true, we see that $\mathbf{proj}_{X_i}(\rho) \in \Phi_i$ for both $i \in \{0,1\}$. □

IV. The Negotiation Process

Our goal is to iteratively compute a pair of compatible contracts. Our procedure will be *sound*: if it returns contracts (A_0, G_0) and (A_1, G_1), we shall be certain that the premises of Thm. III.4 hold. However, since the distributed synthesis problem is undecidable, we may not find compatible contracts.

The iterative computation progresses in rounds. Initially (round 0), the assumptions A_0, A_1 and the guarantees G_0, G_1 allow all behaviors. In each round, C_0 and C_1 check if each can realize the specification $\langle A_i \triangleright \Phi_i \triangleright G_i \rangle$. If so, the iteration ends, and we return the current assumptions and guarantees. On the other hand, if either system surely cannot realize its respective contract, then there is no point continuing and the process stops with failure. If none of the above happens, the negotiation process continues and the systems take turns to refine their assumptions and guarantees.

The key step in refining the assumptions and guarantees requires finding a *sufficient assumption on the other system* that enables realization of the current specification. In principle, this assumption should also be *maximally permissive* to offer maximal freedom to the other system. We first define maximally permissive sufficient assumptions, and then use this definition to formalize the negotiation procedure. At the same time, we also demonstrate that maximal permissiveness comes with its own technical challenges and lack of practicality. So in the end, we resort to a more practical and non-maximally permissive assumption, and show how to compute them.

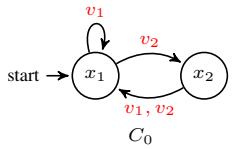

Fig. 3: A simple game

A. Maximally Permissive Sufficient Assumption

We fix an input $(C_0, \Phi_0, C_1, \Phi_1)$. A language $L \subseteq \mathbf{Y_{1-i}}^\omega$ is a *maximally permissive sufficient assumption*—simply *assumption* in short—for Φ_i if (1) L is *sufficient*, i.e., C_i can realize $(L \Rightarrow \Phi_i)$ and (2) L is *maximally permissive*, i.e., C_i cannot realize $(L' \Rightarrow \Phi_i)$ for any proper superset $L' \supsetneq L$.

Intuitively, an assumption is a sufficient restriction on the other system: C_i can realize its specification provided the other system always produces outputs belonging to the assumption. Moreover, the assumption is maximal in that no proper superset is sufficient for C_i to realize its specification: this ensures we restrict the other system in the least pervasive way.

If C_i can already realize Φ_i on its own, then the assumption is all of $\mathbf{Y_{1-i}}^\omega$. If C_i surely cannot realize Φ_i, then $\emptyset$ is a maximal assumption. The maximality constraint rules out "trivial" solutions such as $L = \emptyset$ in other cases.

Example IV.1. *Consider system C_0 in Fig. 3. We assume U_0 is a singleton, and so we have omitted it from the figure. There is a single system strategy π that picks the singleton action at each state. The external variable has values $\{v_1, v_2\}$. Consider the specification $\square\Diamond x_2$, that requires the state x_2 to be visited infinitely often. The system cannot realize this property if from some point on, the environment keeps playing v_1 when the state is x_1. Thus, a maximally permissive sufficient assumption is $L_1 = ((v_1^*(v_2(v_1 + v_2))^*)^* v_2)^\omega$. A sufficient assumption is $L_2 = (v_1 v_2)^\omega$, which is* not *maximally permissive because $L_1 \supsetneq L_2$.*

The following example suggests that maximally permissive sufficient assumptions are not unique.

Example IV.2. *Consider a system with two input actions H and T and an external variable with two values, also called H and T. From the initial state x_{in}, the inputs H and T take the system to two different states h and t, respectively. Once at h or t, the state returns to x_{in} only if the environment plays H or T, respectively, but otherwise the state goes to bad and subsequently stays there no matter what input or action is chosen. Consider the specification $\square\neg bad$. Since the system does not know what the environment will play next, the specification is not realizable. However,* any *singleton ω-language is a maximally permissive sufficient assumption. Thus, there are infinitely many incomparable assumptions, and they even may not be ω-regular.*

Maximality is useful to ensure non-trivial assumptions but, as Ex. IV.2 shows, this may lead to unbounded iterations through strategies and counter-strategies. Moreover, a maximal assumption might leave the system with only a single available realization strategy [15]. To mitigate these two issues, we consider an under-approximation of maximal assumptions. Note that strengthening an assumption retains realizability. A sufficient assumption L for Φ is called *universally* maximally permissive, if (1) L is a sufficient assumption, and (2) L is the intersection of every maximally permissive assumption. Universally maximal assumptions give up completeness—$\emptyset$ is the only universally maximal assumption in Ex. IV.2—but bound the search space of assumptions because it is unique. Moreover, it gives maximal freedom to the system in choosing from all possible realization strategies.

Procedure *FindAssumptions*. For the moment, we assume the following: there is a procedure *FindAssumptions* that given a system C, a specification Φ, and safe ω-languages (A, G), returns a safe sufficient assumption for the specification $\langle A \rhd \Phi \rhd G\rangle$. We expect that the assumption gives as much freedom to the other system as possible. We shall discuss implementations of *FindAssumptions* subsequently.

B. Negotiation

As mentioned earlier, the iterative computation of contracts proceeds in rounds, called *negotiation*. We call the overall iterative algorithm *Negotiate*, and the steps are summarized in Proc. 1. Initially (round 0), the existing assumptions A_0, A_1 and the guarantees G_0, G_1 allow all behaviors:

$$A_0^{(0)} = \mathbf{Y_1}^\omega,\ \ G_0^{(0)} = \mathbf{X_0}^\omega,\ \ A_1^{(0)} = \mathbf{Y_0}^\omega,\ \ G_1^{(0)} = \mathbf{X_1}^\omega.$$

Suppose we have constructed $A_i^{(r)}, G_i^{(r)}$ for $i \in \{0,1\}$ in round r. Let us look at round $r+1$.

In round $r+1$, if either system C_i surely cannot realize the specification $G_i^{(r)} \cap \Phi_i$, we can give up with failure—certainly, we shall not find a refinement that works. In this case Proc. 1 returns **DoesNotExist**. On the other hand, if both systems can realize their specification under the current contracts $\left(A_i^{(r)}, G_i^{(r)}\right)$, we have converged and we can stop and return the current contracts.

If the above conditions do not hold, we first pick the assumption L from *FindAssumptions* $\left(C_0, \left(A_0^{(r)}, G_0^{(r)}\right), \Phi_0\right)$, and if L is non-trivial then update the assumptions and guarantees as follows:

$$A_0^{(r+1)} := A_0^{(r)} \cap L, \quad G_1^{(r+1)} := G_1^{(r)} \cap h_1^{-1}(L). \quad (5)$$

Then, we pick the additional sufficient assumption L' from *FindAssumptions* $\left(C_1, \left(A_1^{(r)}, G_1^{(r+1)}\right), \Phi_1\right)$, and if L' is non-trivial then update the assumptions and guarantees as follows:

$$A_1^{(r+1)} := A_1^{(r)} \cap L', \quad G_0^{(r+1)} := G_0^{(r)} \cap h_0^{-1}(L'). \quad (6)$$

We move to the next round with these new contracts. Note that, at the end of every round, Proc. 1 either fails to find a contract and returns **DoesNotExist**, or else it obtains the sets $A_0^{(r)}$, $A_1^{(r)}$, $G_0^{(r)}$, and $G_1^{(r)}$ which are *all nonempty* as required by Def. III.3.

Procedure 1 *Negotiate*

Input: $(C_0, \Phi_0, C_1, \Phi_1)$
Output: A_0, G_0, A_1, G_1 *or* **DoesNotExist**

$A_0^{(0)} \leftarrow \mathbf{Y_1}^\omega$, $G_0^{(0)} \leftarrow \mathbf{X_0}^\omega$, $A_1^{(0)} \leftarrow \mathbf{Y_0}^\omega$, $G_1^{(0)} \leftarrow \mathbf{X_1}^\omega$
for $r = 0, 1, 2, \ldots$ **do**
  if C_i can realize $\langle A_i^{(r)} \rhd \Phi_i \rhd G_i^{(r)}\rangle$ for both $i \in \{0,1\}$ **then**
    return $A_0^{(r)}, G_0^{(r)}, A_1^{(r)}, G_1^{(r)}$
  end if
  if C_i surely cannot realize $\langle A_i^{(r)} \rhd \Phi_i \rhd G_i^{(r)}\rangle$ for either of $i \in \{0,1\}$ **then**
    return DoesNotExist
  end if
  $L \leftarrow$ *FindAssumptions* $\left(C_0, \left(A_0^{(r)}, G_0^{(r)}\right), \Phi_0\right)$
  $A_0^{(r+1)} := A_0^{(r)} \cap L$, $G_1^{(r+1)} := G_1^{(r)} \cap h_1^{-1}(L)$
  $L' \leftarrow$ *FindAssumptions* $\left(C_1, \left(A_1^{(r)}, G_1^{(r+1)}\right), \Phi_1\right)$
  $A_1^{(r+1)} := A_1^{(r)} \cap L'$, $G_0^{(r+1)} := G_0^{(r)} \cap h_0^{-1}(L')$
end for

We are ready to state our main theorem on *Negotiate*.

Theorem IV.3. *If* $Negotiate(C_0, \Phi_0, C_1, \Phi_1)$ *returns contracts* (A_0, G_0), (A_1, G_1), *then the contracts* (A_0, G_0) *and* (A_1, G_1) *are compatible, and moreover each* C_i *can realize* $\langle A_i \rhd \Phi_i \rhd G_i\rangle$ *for* $i \in \{0,1\}$.

Proof of Thm. IV.3. We show that both the conditions of Def. III.3 are met: Cond. a follows by induction over the round indices r and by the construction of the assumptions and guarantees in each round. (Actually, we maintain the invariant $G_i^{(r)} = h_i^{-1}(A_{1-i}^{(r)})$ at each round r, which is stronger than Cond. a.) Cond. b follows from the condition on successful termination. □

C. Implementing FindAssumptions

We now describe the algorithm *FindAssumptions* to compute a safe under-approximation of the universally maximally permissive sufficient assumption. For algorithmic effectiveness, we only find assumptions and guarantees which are safe *ω-regular languages* (compared to safe ω-languages as per Def. III.2). This restriction allows us to implement *FindAssumptions* by using operations on *finite* structures. Our algorithm uses as subroutines both the (non-distributed) realizability algorithm from [25], [30], [33] and the algorithm to compute environment assumptions from [15].

Subroutine I: Centralized Reactive Synthesis. The following theorem (summarizing [30], [33]) outlines a method that can be used for solving a reactive synthesis problem in the presence of assume-guarantee contracts. Given a system C, a natural number $d > 0$, and a mapping $c : \mathbf{X} \to \{0, \ldots, d\}$ that maps each state to a priority, recall that a *parity objective* $\Psi(c)$ states that the minimum priority visited infinitely often is even.

Theorem IV.4. *Let $C = (X, x_{in}, U, W, \delta, Y, h)$ be a system, and $V \subseteq W$ be a subset of disturbance variables under the control of another system C'. Given a pair of ω-regular languages $A \subseteq \mathbf{V}^\omega$ and $G \subseteq \mathbf{Y}^\omega$, and an ω-regular specification $\Phi \subseteq \mathbf{X}^\omega$, there is an effectively constructible system $\widetilde{C} = (\widetilde{X}, \widetilde{x_{in}}, U, W, \widetilde{\delta}, Y, \widetilde{h})$, a number $d \geq 0$, and a parity specification $\Psi(c)$ for a mapping $c : \widetilde{\mathbf{X}} \to \{0, \ldots, d\}$, such that the following hold:*

(i) System $\widetilde{C}$ can realize $\Psi(c)$ if and only if system C can realize $\langle A \rhd \Phi \rhd G \rangle$.

(ii) System $\widetilde{C}$ can maybe-realize $\Psi(c)$ in the composition $\widetilde{C} \parallel C'$ if and only if system C can maybe-realize $\langle A \rhd \Phi \rhd G \rangle$ in the composition $C \parallel C'$.

(iii) There is a mapping from the set of memoryless realization strategies of the form $\widetilde{\pi} : \widetilde{\mathbf{X}} \to U$ of $\widetilde{C}$ to the set of realization strategies of C.

(iv) There is a mapping from the set of memoryless joint realization strategies of the form $\widetilde{\pi} : \widetilde{\mathbf{X}} \to U \times \mathbf{V}$ of $\widetilde{C}$ and C' to the set of joint realization strategies of C and C'.

The system $\widetilde{C}$ is essentially the product of the original system with a deterministic parity automaton for the specification $\langle A \rhd \Phi \rhd G \rangle$ (see, e.g., [15]). Recall that parity specifications have memoryless realization strategies, and each strategy of $\widetilde{C}$ can be converted to a strategy (possibly using memory) for the original system.

Subroutine II: Finding Environment Assumptions. The second subroutine is the method of Chatterjee et al. [15] to compute a minimal set of sufficient environment assumptions for satisfaction of a given omega-regular specification. At a high-level, their algorithm imposes safety and liveness restriction on sets of environment behaviors which help the system to realize its specification. The safety restrictions, here called *safe-sufficient restrictions*, require that certain environment actions be never applied at certain system states. The liveness restrictions, here called *live-sufficient restrictions*, require that certain environment actions be repeatedly taken if certain system states are repeatedly visited (strong liveness property). We formalize these in the following.

Fix a system composition $C_0 \parallel C_1$. First, we formalize safe-sufficient and live-sufficient restrictions w.r.t. system C_0, with the understanding that similar results can be obtained for C_1 just by changing the indices. Let $\Phi \subseteq (\mathbf{X_0})^\omega$ be a local parity specification for C_0.

A set of pairs $E_s \subseteq \mathbf{X_0} \times \mathbf{Y_1}$ is a *safe-sufficient restriction* on C_1 for Φ if there exists a strategy $\pi : \mathbf{X_0} \times \mathbf{W_0} \to U$ of C_0 such that for every joint counter-strategy $\pi' : \mathbf{X_0} \to \mathbf{W_0}$ of C_1 and the environment, the resulting run $\rho(\pi, \pi') \equiv x^0 \xrightarrow{w^0, u^0} x^1 \xrightarrow{w^1, u^1} x^2 \xrightarrow{w^2, u^2} \ldots$ satisfies the following: either (a) there exists a $i \geq 0$ with $(x^i, w^i[Y_1]) \in E_s$, or (b) for all $i \geq 0$, $x^i \in \langle\langle C_0, C_1 \rangle\rangle \Phi$. A safe-sufficient restriction E_s is *unfair* if there is a run ρ and there is a pair $(x, y) \in E_s$ such that at some time instant j, $\mathbf{proj}_{X_0}(\rho)^j = x$ and $\mathbf{proj}_{Y_1}(\rho)^j = y$, and yet $\rho \in \Phi$. A safe-sufficient restriction is *fair* if it is not unfair, and moreover it is *minimal* if no other safe-sufficient restriction of smaller size exists.

A set of pairs $E_l \subseteq \mathbf{X_0} \times \mathbf{Y_1}$ is a *live-sufficient restriction* on C_1 for Φ if there exists a strategy $\pi : \mathbf{X_0} \to U$ of C_0 such that for every joint counter-strategy $\pi' : \mathbf{X_0} \to \mathbf{W_0}$ of C_1 and the environment, the resulting run $\rho(\pi, \pi')$ satisfies the following: either (a) there exists a pair $(x, y) \in E_l$ such that in the run $\rho(\pi, \pi')$, x appears infinitely often but after some finite time step, y never immediately follows x, or (b) $\mathbf{proj}_{X_0}(\rho(\pi, \pi')) \in \Phi$. A live-sufficient restriction E_l is called *minimal* if no other live-sufficient restriction of smaller size exists. A live-sufficient restriction E_l is called *locally minimal* if no strict subset of E_l is itself a live-sufficient restriction.

The restrictions E_s and E_l for Φ induce a safety assumption $\Psi_{E_s} \subseteq \mathbf{Y_1}^\omega$ and a liveness assumption $\Psi_{E_l} \subseteq \mathbf{Y_1}^\omega$ on the output behavior of the system C_1 respectively. The set Ψ_{E_s} is the set of all infinite output words $y \in \mathbf{Y_1}^\omega$ of C_1 such that there exists a run ρ with $y = \mathbf{proj}_{Y_1}(\rho)$, and for all $i \geq 0$, $(x^i, y^i) \notin E_s$. The set Ψ_{E_l} is the set of all infinite output words $y \in \mathbf{Y_1}^\omega$ of C_1 such that there exists a run ρ with $y = \mathbf{proj}_{Y_1}(\rho)$, and for all $(x^i, y^i) \in E_l$, either x^i appears finitely often in ρ or (x^i, w^i) appears infinitely often in ρ.

The important observations about Ψ_{E_s} and Ψ_{E_l} are summarized in the following theorem.

Theorem IV.5. *[15] For a system composition $C_0 \parallel C_1$, the following assertions hold:*

1) If C_0 can maybe-realize Φ, then there exists a unique minimal fair safe-sufficient restriction E_s on C_1.

2) If (a) Φ is a reachability, safety, or Büchi specification and (b) C_0 can realize the specification $\square\langle\langle C_0, C_1 \rangle\rangle \Phi$, then if there exists a sufficient assumption $\Psi \neq \emptyset$ for Φ, then there exists a live-sufficient restriction E_l on C_1.

3) Let $\Phi_0 \subseteq \mathbf{X_0}^\omega$ be a parity specification, and E_s, E_l be respectively the safe-sufficient and the live-sufficient restrictions on C_1 for Φ. If $\Psi = \Psi_{E_s} \cap \Psi_{E_l} \neq \emptyset$, then Ψ is a sufficient assumption for Φ.

4) The unique minimal fair safe-sufficient restriction E_s can be computed in polynomial time for parity objective Φ, whereas computation of a minimal live-sufficient restriction E_l is NP-hard already for Büchi specifications. There is a polynomial time algorithm for finding a locally minimal live-sufficient restriction E_l for parity specifications.

Implementation. Consider a call to *FindAssumptions* with input C_i, A_i, G_i, and Φ_i. The procedure first checks if C_i can realize the specification $\langle A_i \rhd \Phi_i \rhd G_i \rangle$. If so, it returns the set of all output strings of C_{1-i}: all environment behaviors are allowed.

Otherwise, using Theorem IV.5, *FindAssumptions* computes *the* minimal fair safe-sufficient restriction E_s and *a* locally minimal live-sufficient restriction E_l for the winning condition $\langle \text{True} \rhd \Phi_i \rhd G_i \rangle$. The reason we replaced A_i with True in this case is to avoid the trivial case when C_i realizes $\langle A_i \rhd \Phi_i \rhd G_i \rangle$ with the contradictory assumption that demands A_i be violated.

Next, *FindAssumptions* under-approximates the liveness assumption Ψ_{E_l} by a safety language $\Psi_{E_{l \to s}}$ as outlined in the following. First we introduce some notation. For the live-sufficient restriction E_l and for a given state $x \in \mathbf{X_i}$, we use

the notation $E_l(x)$ to denote the set $\{y \in \mathbf{Y_i} \mid (x,y) \in E_l\}$. We assume that the elements of the set E_l has been assigned some index, and we use the notation $E_l(x)_i$ to denote the i-th element of the set $E_l(x)$. We use the notation $|E_l(x)|$ to denote the cardinality of the set $E_l(x)$. The set $\Psi_{E_{l\to s}}$ is the set of all output words $w \in \mathbf{Y_{1-i}}^\omega$ produced by C_{1-i} which satisfy the following: (a) there exists a run ρ (for some set of strategies) of C_i with $w = \mathbf{proj}_{Y_{1-i}}(\rho)$, and (b) for all $x \in \operatorname{dom} E_l$, every j-th occurrence of x in the projection $\mathbf{proj}_{X_i}(\rho)$ should be immediately followed by $E_l(x)_k$ where $k = j \mod |E_l(x)|$. The procedure $FindAssumptions$ returns the safety language $\Psi_{E_s} \cap \Psi_{E_{l\to s}}$.

In the following theorem, we formally state the properties of the procedure $FindAssumptions$.

Theorem IV.6. *Let the language returned by the procedure $FindAssumptions(C, A, G, \Phi)$ be Ψ. The following assertions hold:*

1) *The language Ψ is a sufficient assumption for the specification $\langle A \rhd \Phi \rhd G\rangle$.*
2) *When $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$ is a safety language, Ψ is a universally maximally permissive and sufficient assumption for the specification $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$.*

Proof. (1) Suppose $FindAssumptions$ returns the language $\Psi := \Psi_{E_s} \cap \Psi_{E_{l\to s}}$. Let $\Psi' := \Psi_{E_s} \cap \Psi_{E_l}$. First we show that Ψ is a sufficient assumption for the specification $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$. By Thm. IV.5.3, we have that Ψ' is a sufficient assumption for realizing the specification $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$. We show that $\Psi_{E_{l\to s}} \subseteq \Psi_{E_l}$, which would imply that $\Psi \subseteq \Psi'$, which in turn would establish that Ψ is also a sufficient assumption. For every valid run ρ of the underlying game graph belonging to the set $\Psi_{E_{l\to s}}$, for *every* visit of a state $s \in \operatorname{dom} E_l$, the environment chooses edges from E_l in a round-robin fashion. This trivially implies that the strong fairness condition in Ψ_{E_l} is satisfied by ρ, and hence $\rho \in \Psi_{E_l}$. This proves the claim $\Psi_{E_{l\to s}} \subseteq \Psi_{E_l}$, and it is established that Ψ is a sufficient assumption for $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$.

Now we show that Ψ is also a sufficient assumption for the specification $\langle A \rhd \Phi \rhd G\rangle$. There are either of the following two ways that $\langle A \rhd \Phi \rhd G\rangle$ can be satisfied by the system:
(i) The specification $\Phi \wedge G \equiv \langle \mathsf{True} \rhd \Phi \rhd G\rangle$ always holds.
(ii) There exists a finite run $r \equiv x^0 \xrightarrow{w^0,u^0} x^1 \to \ldots x^k$ compliant with π, such that $\mathbf{proj}_X(r) \in \operatorname{pref}(G)$, and $\mathbf{proj}_V(r) \notin \operatorname{pref}(A)$.
This shows that $\langle A \rhd \Phi \rhd G\rangle$ is a weakening of the specification $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$ for any $\emptyset \subsetneq A \subseteq \mathbf{Y_i}^\omega$. Since the component can realize $\Psi \Rightarrow \langle \mathsf{True} \rhd \Phi \rhd G\rangle$, hence it can also realize $\Psi \Rightarrow \langle A \rhd \Phi \rhd G\rangle$. Thus, Ψ is indeed a sufficient assumption for the specification $\langle A \rhd \Phi \rhd G\rangle$.
(2) When $\langle \mathsf{True} \rhd \Phi \rhd G\rangle$ is a safety language, then $E_l = \emptyset$ and as a result $\Psi = \Psi_{E_s}$. The rest follows from the fact that E_s is the minimal fair safe-sufficient restriction . □

Note that, due to the non-uniqueness of the locally minimal live-sufficient restriction and the non-uniqueness of the ordering that we impose on the elements of the set $E_l(x)$ for every x, the assumption computed by $FindAssumptions$ is in general not unique. Nevertheless, owing to the finiteness of the systems there are only *finitely* many possible ways to choose both of these, and as a result the number of assumptions is also *finite*. This creates the possibility of extending our basic negotiation procedure to backtrack and retry with a different set of assumptions upon failure. Naturally, this *finitely branching negotiation* procedure will be relatively more "complete" than the non-branching procedure $Negotiate$. However, for a cleaner exposition of the main theory and as a first step, in this paper we only focus on the non-branching version with a focus on "soundness", and plan to work in future on the branching version with a focus on "relative completeness".

V. Implementation and Approximations

We have built a prototype C++-based tool called Agnes that implements the negotiation algorithm. Agnes is freely available at https://github.com/kmallik/Agnes. The name Agnes stands for **A**ssume-**G**uarantee **NE**gotiation for distributed **S**ynthesis. Agnes accepts descriptions of the systems and the local specifications in a list representation (list of states, list of transitions, etc.) given as text files. At the moment, the tool only supports safety and deterministic Büchi conditions as local specifications. If the negotiation is successful, Agnes outputs contracts where the guarantees (same as assumptions of the other systems) are modeled as finite automata, also stored using a list representation or in DOT format for visualization.

We have implemented a number of heuristics on top the general algorithm for better performance. We describe the main ones below.

A. Pattern-based Under-approximation of Assumptions

Because the construction of Thm. IV.4 depends on the sizes of these automata, we now discuss a heuristic that tries to find small automata. Observe that if C realizes $\langle A \rhd \Phi \rhd G\rangle$, then it also realizes $\langle A' \rhd \Phi \rhd G\rangle$ for any $A' \subseteq A$. Thus, we heuristically find stronger assumptions that can be implemented by smaller automata. The intuition behind our heuristic is that we assume bad behaviors of the environment (those that do not satisfy the safety assumption) can be identified by a set of short unsafe suffixes: as long as the environment does not produce these short suffixes, the assumption continues to hold. It is inspired by the notion of l-completeness in control [27], which abstracts a system by only tracking the states visited in the last l steps.

Our heuristic works as follows. First, we note that, because the assumption is a safety language, the $FindAssumptions$ procedure returns a *universal* Büchi automaton over words for a prefix-closed ω-language. Since the language is prefix-closed, the automaton has a single rejecting sink state, and all other states are accepting. We can dualize this automaton to get a non-deterministic Büchi automaton for the negation of the language. In the negation, the sink state is the only accepting state. Let us call this automaton $\mathcal{A}$.

Given $\mathcal{A}$ and a (user-supplied) parameter k, we now construct an automaton $\mathcal{B}$ that accepts a superset of the language of $\mathcal{A}$. The automaton $\mathcal{B}$ keeps all states of $\mathcal{A}$ that have some path of length at most k to the unique accepting state and merges all states for which the shortest path to the

accepting state is greater than k. Formally, $Q_{\mathcal{B}} = \{q \in Q_{\mathcal{A}} \mid \exists$ path from q to the accepting state of length $\leq k\} \cup \{r\}$, for a new state r. Consider a mapping $\lambda : Q_{\mathcal{A}} \to Q_{\mathcal{B}}$ that maps the subset $Q_{\mathcal{B}} \setminus \{r\}$ identically to itself and maps every other state to r. The transitions of $\mathcal{B}$ consist of transitions $(\lambda(q), a, \lambda(q'))$ for each transition (q, a, q') in $\mathcal{A}$. The initial and final states of $\mathcal{B}$ are the map of the initial and final states of $\mathcal{A}$ under λ.

Clearly, the number of states in $\mathcal{B}$ is less than or equal to that of $\mathcal{A}$ and $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Finally, we dualize $\mathcal{B}$ to get back a universal Büchi automaton that accepts the new assumption which is contained in the original assumption.

While this heuristic that the bad environment behaviors can be identified by short suffixes might not work in general, it worked well in our examples.

B. Büchi Specifications

We now present an optimization that is orthogonal to the one presented in Sec. V-A. Recall that the procedure *FindAssumptions* requires computation of locally minimal live-sufficient restriction, for which we used the algorithm proposed by Chatterjee et al. [15]. Here we present a greedy algorithm for the same purpose, but for the special case when the specification of the system $\widetilde{C}$ in Thm. IV.4 can be represented as a Büchi condition $\square\lozenge B$ for some $B \subseteq \widetilde{\mathbf{X}}$. Already for this case, computing the minimal live-sufficient restriction is NP-hard [15, Thm. 11]. The algorithm presented by Chatterjee et al. [15] to compute a *locally* minimal live-sufficient restriction takes $\mathcal{O}(n^6)$ time, where n is the size of the state space. Our greedy algorithm runs in time $\mathcal{O}(n^3)$.

We introduce some notation before presenting our algorithm. Fix a composition $C_0 \parallel C_1$ and a specification $\langle A_0 \rhd \Phi_0 \rhd G_0 \rangle$ for the system C_0. Consider a Büchi specification $\square\lozenge B$ for some $B \subseteq \widetilde{\mathbf{X_0}}$, where $\widetilde{X_0}$ is the state space of the product system $\widetilde{C_0}$ as defined in Thm. IV.4. Let E_l be a live-sufficient restriction, and $AssumeFair(E_l, \square\lozenge B)$ be the set of infinite sequences of states $x_0 x_1 \ldots$ s.t. there exists a strategy π of C_0 and a joint strategy π' of C_1 and the environment s.t. $\mathbf{proj}_{\widetilde{X}_0}(\rho(\pi, \pi')) = x_0 x_1 \ldots$ and $x_0 = \widetilde{x_{in0}}$, and moreover either (a) there exists a pair $(x, w) \in E_l$ s.t. in the run $\rho(\pi, \pi')$, x appears infinitely often but after some finite time step, w never immediately follows x, or (b) $\mathbf{proj}_{\widetilde{X}_0}(\rho(\pi, \pi')) \in \Phi$.

Proc. 2 uses a greedy algorithm to compute a live-sufficient restriction E_l: the algorithm progressively expands the realizable region for $\lozenge B$ by greedily adding all the favorable restrictions on C_1 to E_l whenever needed.

The heuristic does not generalize to other ω-regular specifications. Recall that in μ-calculus notation, the Büchi fixpoint is written as follows [25]:

$$Z^* = \nu Z \,.\, \mu Y \,.\, \mathrm{Cpre}(Y) \vee (\mathrm{Cpre}(Z) \wedge B),$$

where $\mathrm{Cpre} : 2^{\mathbf{X}} \to 2^{\mathbf{X}}$, $\mathrm{Cpre} : S \mapsto \{x \in \mathbf{X} \mid \exists u \in U \,.\, \forall w \in \mathbf{W} \,.\, \delta(x, w, u) \in S\}$ is the controllable predecessor operator, and Z^* is the final Büchi winning region. In Proc. 2, we exploited the fact that when C_1 co-operates with C_0, Z^* is a priori known to be the set $\langle\langle C_0, C_1 \rangle\rangle \square\lozenge B$. Then the problem gets simplified to finding a locally minimal live-sufficient restriction E_l s.t. C_0 can (independently) realize the conditional

Procedure 2 Compute a live-sufficient restriction E_l

Input: System C_0 in the composition $C_0 \parallel C_1$, Büchi states $B \subseteq X_0$

Output: A live-sufficient restriction E_l on C_1, or **DoesNotExist**

1: **if** $x_{in0} \notin \langle\langle C_0, C_1 \rangle\rangle \square\lozenge B$ **then**
2: **return DoesNotExist**
3: **end if**
4: $Target \leftarrow B \cap \langle\langle C_0, C_1 \rangle\rangle \square\lozenge B$
5: $WinDom \leftarrow \langle\langle C_0 \rangle\rangle \lozenge Target$
6: $E_l \leftarrow \emptyset$
7: **while** $AssumeFair(E_l, \square\lozenge B)$ is not realizable **do**
8: $E_l \leftarrow E_l \cup \{(x, y) \in (\mathbf{X_0} \times \mathbf{Y_1}) \mid x \notin WinDom \wedge \exists u \in U_0 \,.\, \forall w \in \mathbf{W_0} \setminus \mathbf{Y_1} \,.\, \delta_0(x, (w, y), u) \in WinDom \wedge \exists y' \in \mathbf{Y_1} \,.\, \forall u \in U_0 \,.\, \exists w \in \mathbf{W_0} \setminus \mathbf{Y_1} \,.\, \delta_0(x, (w, y'), u) \notin WinDom\}$
9: $Target \leftarrow WinDom \cup \mathrm{dom}\, E_l$
10: $WinDom \leftarrow \langle\langle C_0 \rangle\rangle \lozenge Target$
11: **end while**
12: **return** E_l

reachability problem $AssumeFair(E_l, \lozenge(\mathrm{Cpre}(Z^*) \cap B))$. We solve this problem by iterating through a growing sequence of E_l, where in each iteration we add those environment behaviors to E_l which would immediately help to grow the winning region for $\lozenge(\mathrm{Cpre}(Z^*) \cap B)$. While this heuristic works well for the Büchi specification, it will not work so well for every other ω-regular specification. Already for co-Büchi specifications, where the order of the "μ" and "ν" iterations gets reversed, this heuristic is not suitable.

VI. Experimental Evaluation

A. Notation

Before summarizing the experimental results, let us introduce some notation. Given a system C_i and a contract (A_i, G_i), we use $|G_i|$ to represent the size of the state space of the universal Büchi automaton which accepts the language G_i. We use the parameter k to represent the user-supplied parameter used to determine the level of minimization used for minimizing the size of the contracts (see Sec. V-A): we start with $k = 1$ and then keep increasing the value of k until a pair of compatible contracts is found, or until a point when we realize that increasing k any more would not change the outcome, whichever happens earlier. We use the status flag **S** and **F** to represent these two situations respectively. Indeed, it was found while inspecting the examples that the cases with status **F** do not have a contract that can be represented using a safe under-approximation of the universally maximally permissive assumptions. We put a cross mark ("×") for the entries $|G_0|$ and $|G_1|$ in all the failed cases as they are irrelevant.

The experimental results are going to be summarized in Table I and Table II. The key highlight in the tables is that the pattern-based minimization of the assumptions (see Sec. V-A) turns out to be extremely beneficial while performing the

TABLE I: Experimental evaluation of the distributed packet sending problem.

l_0, p_0, t_0	l_1, p_1, t_1	$\|X_0\|$	$\|X_1\|$	No pattern-based opt		Pattern-based optimization (Sec. V-A)											
						$k=1$		$k=2$		$k=3$		$k=4$		$k=5$			
				time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status		
				$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$
$(1,1,1)$	$(0,1,1)$	5	3	< 0.001	**F**	0.001	**F**										
				×	×	×	×										
$(1,1,2)$	$(0,1,1)$	7	3	0.001	**S**	0.001	**S**										
				2	4	2	2										
$(1,1,2)$	$(1,1,3)$	7	9	0.004	**S**	0.001	**F**	< 0.001	**F**	0.003	**S**						
				5	4	2	2	2	3	4	4						
$(2,2,4)$	$(1,1,3)$	35	9	0.018	**S**	< 0.001	**F**	0.001	**F**	0.010	**S**						
				5	8	2	2	2	5	4	6						
$(2,2,5)$	$(1,1,3)$	43	9	0.112	**S**	0.002	**F**	0.002	**F**	0.027	**S**						
				8	11	2	2	2	5	4	6						
$(2,2,5)$	$(2,2,5)$	43	43	0.085	**S**	0.001	**F**	0.002	**F**	0.003	**F**	0.066	**S**				
				6	11	2	2	2	5	2	6	5	11				
$(3,3,14)$	$(2,2,8)$	255	67	18.996	**S**	0.006	**F**	0.007	**F**	0.011	**F**	0.312	**S**				
				40	99	2	2	2	6	2	16	5	17				
$(4,3,14)$	$(3,2,8)$	339	99	42.254	**S**	0.007	**F**	0.010	**F**	0.023	**F**	0.027	**F**	14.967	**S**		
				47	129	2	2	2	6	2	22	2	23	18	129		

TABLE II: Experimental evaluation of the tandem queuing network example.

t_0	p, t_1	$\|X_0\|$	$\|X_1\|$	No pattern-based opt		Pattern-based optimization (Sec. V-A)															
						$k=1$		$k=2$		$k=3$		$k=4$		$k=5$		$k=6$		$k=7$			
				time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status	time(s)	status		
				$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$	$\|G_0\|$	$\|G_1\|$
3	$(1,1)$	8	4	0.055	**S**	0.002	**F**	0.002	**F**	0.006	**F**	0.055	**S**								
				15	6	2	2	2	3	5	4	15	6								
3	$(2,1)$	8	6	0.073	**S**	0.002	**F**	0.002	**F**	0.008	**F**	0.076	**S**								
				17	6	2	2	2	3	5	4	17	6								
3	$(2,2)$	8	8	0.008	**F**	0.027	**F**														
				×	×	×	×														
4	$(2,2)$	10	8	0.262	**S**	0.003	**F**	0.003	**F**	0.006	**F**	0.031	**S**								
				32	9	2	2	2	3	2	4	7	5								
4	$(3,2)$	10	10	0.019	**F**	0.079	**F**														
				×	×	×	×														
5	$(3,2)$	12	8	1.076	**S**	0.003	**F**	0.003	**F**	0.007	**F**	0.013	**F**	0.065	**S**						
				62	13	2	2	2	3	2	4	2	5	9	6						
5	$(3,3)$	12	12	0.030	**F**	0.109	**F**														
				×	×	×	×														
7	$(3,3)$	16	12	1.560	**S**	0.004	**F**	0.004	**F**	0.008	**F**	0.010	**F**	0.013	**F**	1.456	**F**	0.150	**S**		
				13	8	2	2	2	3	2	4	2	5	2	6	48	26	13	8		

negotiation. In the table, the red cells show the computation time when this optimization was disabled, whereas the blue cells show the computation time for the smallest value of k for which the negotiation was successful. It can be observed that as the systems' state spaces get larger, the saving gets higher. Also, observe the difference in the sizes of the contracts: when this optimization is disabled, the contract sizes (given by $|G_0|$ and $|G_1|$) tend to be much higher.

B. A Distributed Packet Sending Problem

Our first example is a parameterized and scaled up version of the distributed packet sending problem introduced in Sec. II. The parameters for the system C_i with $i \in \{0, 1\}$ are given by: 1) Number of packets to be sent l_i, 2) maximum delay between two consecutive packet transmissions p_i, and 3) the overall time limit to send all the packets t_i. Essentially the only difference with the additional parameters l_i and p_i is that there are two additional counters in the state space to keep track of the respective constraints. In addition to the state $done$, there is one more special *sink* state called $excessive\text{-}delay$ to mark the event that the elapsed time between two consecutive transmissions exceeded the allowed bound p_i. Then the local specification Φ_i for each system can be formalized as $\Diamond(s = done)$. Table I summarizes the experimental results.

C. A Distributed Tandem Queuing Network Problem

Our second example is a tandem queuing network similar to the one used in [21]. Suppose there is a shared queue of *bounded size*. There is a system C_0 that pushes objects to the queue from one end, and there is a system C_1 that pops an object at a time from the other end for processing. System C_0 can only sense if the queue is full or not, has two control actions $push$ and $wait_0$, and produces two outputs $busy$ and $idle_0$ which represent whether C_0 pushed or waited in the last cycle respectively. If the queue is full then C_0 is forced to wait until C_1 draws the next object. On the other hand, system C_1 can only sense if the queue is empty or not, has three control actions $draw$, $wait_1$, and $process$, and produces three outputs $loaded$, $idle_1$, and $processing$ which represent whether C_1 drew an object, waited, or processed an object in the last clock cycle respectively. If the queue is empty, then C_1 is forced to stay *idle* until an object appears in the queue. Let p be some given positive parameter. If the queue is not empty, then C_1 can draw an object, in which case it has to

perform p number of actions on the object before being able to draw another one.

Note that none of the processes has the exact information of the size of the queue. As a result they cannot certainly predict if the queue is going to be full/empty in the next step just by observing their own and the other component's output sequences. We model this uncertainty by introducing an imaginary *environment* player, who can control when the queue is full and when it is empty in an unpredictable manner.

Let t_0 and t_1 be two positive parameters. We introduce two special events for formalizing the specification: (a) System C_0 goes to the $shut\text{-}down$ state if it remains idle for t_0 consecutive time steps and (b) system C_1 goes to the $cool\text{-}down$ state if it remains idle for t_1 consecutive time steps. The local specifications Φ_0 and Φ_1 of C_0 and C_1 are respectively: $\Phi_0 = \Box\neg shut\text{-}down$ and $\Phi_1 = \Box\Diamond cool\text{-}down$. Table II summarizes the experimental results.

VII. Future Work and Conclusion

We have presented a sound procedure for the distributed reactive synthesis problem with two communicating systems. The outcome of our procedure is a pair of safe assume-guarantee contracts. The contracts decompose the synthesis problem: each system realizes its own local specification under the contract in a modular fashion. We model the contract synthesis procedure as a *negotiation* process; the two systems take turns on computing a minimal set of additional favorable behaviors that should be ensured by the other system.

We have presented our procedure for *two* communicating systems. However, as our method generates assume-guarantee contracts which do not require synchronization, it can be generalized ta an arbitrary number of communicating systems. In that case, instead of going back-and-forth between two systems, each iteration of the negotiation will have to consider one system against all the other "neighbors". The collective assumption generated on the neighbors will then need to be projected onto the neighbors' output alphabets. The negotiation can proceed in a round-robin fashion while the systems take turns to propose their assumptions, until all of the systems can realize their contracts.

We presented a prototype implementation of our method that supports safety and deterministic Büchi specifications. We are planning to extend this tool to synthesize contracts for larger examples with general ω-regular specifications.

In some situations *live* contracts are the appropriate choice, as opposed to the safe contracts that we consider in this paper. We leave this generalization to future work.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
[2] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods Syst. Des.*, 15(1):7–48, 1999.
[3] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*, pages 26–33. IEEE, 2013.
[4] R. Alur, S. Moarref, and U. Topcu. Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In *TACAS*, volume 9035 of *LNCS*, pages 501–516. Springer, 2015.
[5] R. Alur, S. Moarref, and U. Topcu. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Inf. Comput.*, 261:616–633, 2018. doi:10.1016/j.ic.2018.02.021.
[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.
[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A.L. Sangiovanni-Vincentelli, W. Damm, T.A. Henzinger, and K.G. Larsen. Contracts for system design. *Found. Trends in EDA*, 12(2-3):124–400, 2018.
[8] R. Bloem, K. Chatterjee, T.A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, volume 5643 of *LNCS*, pages 140–156. Springer, 2009.
[9] R. Bloem, K. Chatterjee, S. Jacobs, and R. Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *TACAS*, pages 517–532. Springer, 2015.
[10] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer. How to handle assumptions in synthesis. *SYNT*, 2014.
[11] R. Bloem, R. Ehlers, and R. Könighofer. Cooperative reactive synthesis. In *ATVA*, pages 394–410, 2015.
[12] R. Brenguier, J.-F. Raskin, and O. Sankur. Assume-admissible synthesis. *Acta Informatica*, 54(1):41–83, Feb 2017.
[13] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.
[14] K. Chatterjee and T.A Henzinger. Assume-guarantee synthesis. In *TACAS*, pages 261–275. Springer, 2007.
[15] K. Chatterjee, T.A. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *CONCUR*, pages 147–161. Springer, 2008.
[16] W. Damm and B. Finkbeiner. Automatic compositional synthesis of distributed systems. In *International Symposium on Formal Methods*, pages 179–193. Springer, 2014.
[17] E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, pages 368–377. IEEE, 1991.
[18] B. Finkbeiner, M. Gieseking, and E.-R. Olderog. Adam: Causality-based synthesis of distributed systems. In *Computer Aided Verification (CAV 15)*, pages 433–439. Springer, 2015.
[19] B. Finkbeiner and S. Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013.
[20] J. Hecking-Harbusch and N.O. Metzger. Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 369–386. Springer, 2019.
[21] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model-checking Markov chains. *STTT*, 4(2):153–172, 2003.
[22] U. Klein and A. Pnueli. Revisiting synthesis of GR(1) specifications. In *HVC*, volume 6504 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2010.
[23] O. Kupferman, G. Perelli, and M.Y. Vardi. Synthesis with rational environments. *Ann. Math. Artif. Intell.*, 78(1):3–20, 2016.
[24] R. Majumdar, N. Piterman, and A.-K. Schmuck. Environmentally-friendly GR(1) synthesis. In *TACAS*, pages 229–246, 2019.
[25] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer, 1995.
[26] K.L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, pages 342–346. Springer, 1999.
[27] T. Moor, J. Raisch, and S. O'Young. Discrete supervisory control of hybrid systems based on l-complete approximations. *Discrete Event Dynamic Systems*, 12(1):83–107, 2002.
[28] K.S. Namjoshi and R.J. Trefler. On the completeness of compositional reasoning methods. *ACM Trans. Comput. Log.*, 11(3):16:1–16:22, 2010.
[29] N. Ozay, U. Topcu, and R.M. Murray. Distributed power allocation for vehicle management systems. In *CDC and ECC*, pages 4841–4848. IEEE, 2011.
[30] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *SIGPLAN-SIGACT*, pages 179–190. ACM, 1989.
[31] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE, 1990.
[32] A. Saoud, A. Girard, and L. Fribourg. On the composition of discrete and continuous-time assume-guarantee contracts for invariance. In *ECC*, pages 435–440. IEEE, 2018.
[33] W. Thomas. On the synthesis of strategies in infinite games. In *STACS'95*, pages 1–13. Springer, 1995.