Synthesis of Coordination Programs from Linear Temporal Specifications

SUGUMAN BANSAL, Rice University, USA
KEDAR S. NAMJOSHI, Nokia Bell Labs, USA
YANIV SA'AR, Nokia Bell Labs, Israel

This paper presents a method for synthesizing a reactive program to coordinate the actions of a group of other reactive programs so that the combined system satisfies a temporal specification of its desired long-term behavior. Traditionally, reactive synthesis has been applied to the construction of a stateful hardware circuit. This work is motivated by applications to other domains, such as the IoT (the Internet of Things) and robotics, where it is necessary to coordinate the actions of multiple sensors, devices, and robots to carry out a task. The mathematical model represents each agent as a process in Hoare's CSP model. Given a network of interacting agents, called an *environment*, and a temporal specification of long-term behavior, the synthesis method constructs a *coordinator* process (if one exists) that guides the actions of the environment agents so that the combined system is deadlock-free and satisfies the given specification. The main technical challenge is that a coordinator may have only *partial information* of the environment state, due to non-determinism within the environment and internal environment actions that are hidden from the coordinator. This is the first method to handle both sources of partial information and to do so for arbitrary linear temporal logic specifications. It is established that the coordination synthesis problem is PSPACE-hard in the size of the environment. A prototype implementation is able to synthesize compact solutions for a number of coordination problems.

CCS Concepts: • **Theory of computation → Concurrency**; **Formal languages and automata theory**; • **Software and its engineering → Correctness**.

Additional Key Words and Phrases: coordination, synthesis, temporal logic

ACM Reference Format:
Suguman Bansal, Kedar S. Namjoshi, and Yaniv Sa'ar. 2020. Synthesis of Coordination Programs from Linear Temporal Specifications. *Proc. ACM Program. Lang.* 4, POPL, Article 54 (January 2020), 27 pages. https://doi.org/10.1145/3371122

1 INTRODUCTION

Coordination problems arise naturally in many settings. In a so-called "smart" building, various sensors, heating and cooling devices must work in concert to maintain comfortable conditions. In a fully automated factory, robots with specialized capabilities must collaborate to carry out manufacturing tasks. Typically, the individual agents are reactive and a centralized coordinator provides the necessary overall guidance to carry out a task.

A coordination program must work in the presence of several complicating factors such as concurrency, asynchrony, and distribution; it should recover gracefully from agent failures and

Authors' addresses: Suguman Bansal, Rice University, Houston, TX, 77005-1892, USA, suguman@rice.edu; Kedar S. Namjoshi, Nokia Bell Labs, Murray Hill, NJ, 07974, USA, kedar.namjoshi@nokia-bell-labs.com; Yaniv Sa'ar, Nokia Bell Labs, Kfar Saba, 4464321, Israel, yaniv.saar.mail@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
2475-1421/2020/1-ART54
https://doi.org/10.1145/3371122

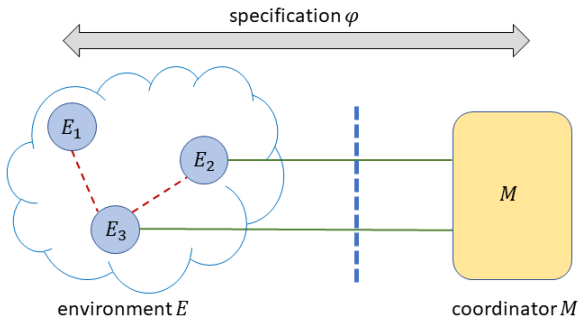

Fig. 1. The coordination model. The coordinator M communicates with the agents $\{E_i\}$ through an interface (shown as solid green lines); it need not have a direct link to every agent. Agent-to-agent interactions (shown as dashed red lines) are hidden from the coordinator, as are actions internal to an agent (not shown). The specification φ is "all-seeing" - it may refer to any action.

handle noisy sensor data. All this complicates the design of coordination programs. It is often the case, however, that the task itself can be specified easily and compactly. We therefore consider whether it is possible to *automatically synthesize* a reactive coordination program from a description of the agents and a specification of the desired long-term system behavior.

We formulate the coordination problem, illustrated in Fig. 1, within the extensively studied framework of Communicating Sequential Processes (CSP) [Hoare 1978, 1985]. Agents and the coordinator are represented as CSP processes. Each process has a set of *private actions*, which are special to the process, and a set of *public actions*, which may be shared with other processes. The *coordination synthesis problem* considered here is defined as follows: given CSP processes $E_1, \ldots, E_n$, each modeling a reactive agent, and a temporal specification φ over their public and private actions, construct a coordinator CSP process M over the interface actions such that all computations of the combined system satisfy the specification φ. The agents are grouped together into a single *environment* process E, formed by the parallel composition of the individual processes $E_1, \ldots, E_n$. An instance of the problem is *realizable* if there is such a CSP process M; otherwise, the given problem instance is *unrealizable*. The temporal specification is expressed in linear temporal logic (LTL) [Pnueli 1977] or, more generally, as an automaton.

This formulation differs from the literature on reactive synthesis in several important ways. A major difference is the choice of CSP as a modeling language. Nearly all of the prior work is based on a *synchronous, shared-variable* model of computation (cf. [Alur et al. 2016; Bloem et al. 2012; Bohy et al. 2012; Büchi and Landweber 1969; Church 1957, 1963; Finkbeiner and Schewe 2013; Kupferman and Vardi 2005; Moarref and Kress-Gazit 2018; Pnueli and Rosner 1989a; Rabin 1969]). That is an appropriate model for hardware design but not for the coordination scenarios described above, simply because those systems are naturally asynchronous: there is no common clock. Pnueli and Rosner [Pnueli and Rosner 1989b] formulate an *asynchronous* shared-variable model, but that is based on a highly adversarial scheduler and is quite weak as a result: for instance, the requirement that an input data stream is copied faithfully to the output has no solution in that model, while it has a simple solution in CSP.

A second difference is the communication model. In the prior work, communication is modeled by reads and writes to a shared memory. This is not a good fit for the targeted application domains, where the natural mode of communication is message passing. For example, a coordinator may send a message to a robot asking it to lift up its arm, or to a toaster to shut it off. Implementing

such handshakes in shared memory requires a dedicated synchronization protocol, complicating modeling as well as synthesis. On the other hand, such communication is easily and directly expressed in CSP, at a higher level of atomicity that permits more specifications to be realized, as illustrated by the data-copying example.

The new formulation crucially differs from prior work on synthesis for CSP [Ciolek et al. 2017; D'Ippolito et al. 2013; Manna and Wolper 1981; Wolper 1982] in its treatment of hidden actions. The prior methods require that *every* environment action is visible to a coordinator. That can result in unnecessarily complex coordinators which are forced to respond to every action, even if not all of those actions are relevant to achieving a goal. Moreover, full visibility goes against the key principle of information hiding in software design. Indeed, the CSP language includes operators to limit the scope and visibility of actions, making models with hidden actions the common case.

A coordinator must, therefore, operate correctly with only *partial information* about the state of the environment agents. First, the state of an agent may be changed autonomously by a private action. Such transitions are hidden from the coordinator, introducing one form of partial information. Second, agents may interact and exchange information through a shared public action; such interactions are also unobserved by the coordinator, introducing yet another form of partial information. Finally, any interaction – including one between an agent and the coordinator – may have a non-deterministic effect, so that the precise next state of the agent is not known to the coordinator, introducing a third form of partial information.

The specification, on the other hand, is "all-seeing"; it may refer to any action of the combined system. For instance, a specification may include a fairness assumption that rules out an uninterrupted sequence of agent-to-agent interactions.

Models with hidden actions alter the synthesis problem in fundamental ways, requiring the development of new algorithms. In the absence of hidden actions, the environment and the coordinator are synchronized. If the coordinator takes n steps, the environment must also have taken n steps. The inclusion of hidden actions introduces *asynchrony*: if the coordinator has taken n steps, the environment may have taken $n + m$ steps, where m is the number of hidden actions; m is unknown to the coordinator and could be unbounded. Although the m hidden actions are invisible to the coordinator, they could be referenced in the specification and cannot simply be ignored. The heart of our algorithm is a transformation that incorporates the effects of the hidden actions into the original temporal specification, producing a new specification that is expressed purely in terms of the interface actions. The transformed specification is then synthesized using existing synthesis methods.

Our work handles specifications written in LTL and carries out the specification transformation using automata-theoretic methods. The transformed automaton has a number of states that is *linear* in the size of the automaton for the negated LTL specification and in the size of the environment model. We show how to express the transformation fully symbolically, which is important as the number of transitions is exponential in the number of interface actions. This is in contrast to prior algorithms for CSP synthesis [Ciolek et al. 2017], which require explicit determinization steps that may result in an *exponential* blowup in the number of states. The final synthesis step using the transformed specification is also carried out symbolically, via a translation to a Boolean constraint problem.

It is necessary to use a full LTL synthesis algorithm for the final step. The GR(1) subset of LTL is often used for specification, as it has efficient symbolic synthesis algorithms for the synchronous model [Bloem et al. 2012; Piterman et al. 2006]. The efficiency advantage is unfortunately lost under asynchrony, as it is not known whether GR(1) is closed under the specification transformation. Thus, even if the original specification is in GR(1), the transformed specification may lie outside GR(1), requiring the use of a general LTL synthesizer.

Coordination synthesis has a high inherent complexity: we show that the question is PSPACE-hard in the size of E for a fixed specification; in comparison, model-checking E is in NLOGSPACE. This rather severe hardness result is not entirely unexpected as problems in reactive synthesis typically exhibit high worst-case complexity. For instance, synthesis in the Pnueli-Rosner model is 2-EXPTIME-complete in the size of the LTL specification [Pnueli and Rosner 1989a,b].

A prototype implementation of our algorithm successfully synthesizes solutions to several coordination problems. It uses Ordered Binary Decision Diagrams (OBDDs) [Bryant 1986] for symbolic manipulation. The transformed specification is then checked for realizability using the state-of-the-art symbolic synthesis tool BoSy [Faymonville et al. 2017b]. We present case studies that synthesize a smart thermostat and an arbiter for a number of concurrent processes. These studies illustrate the capabilities of the model and show how synthesis improves the experience of designing coordination programs. The prototype has limited scalability; the primary bottleneck being the capacity of the back-end synthesis tools. Improvements to these tools (an active research topic) will therefore have a positive effect on the scalability of coordination synthesis. The most impressive impact on scalability is likely to come, however, from entirely new methods for synthesis that use symmetry, modularity, and abstraction strategies to effectively handle large environment state spaces, drawing inspiration from the success of such techniques in formal verification.

2 ILLUSTRATIVE EXAMPLES

We present a series of small examples to illustrate the important features of the model and the considerations that must go into the design of a synthesis algorithm. This section focuses on giving a reader an intuitive view of the issues, precise definitions follow in later sections.

CSP notation and semantics. The full CSP language has a rich structure (cf. [Hoare 1985; Roscoe 1997]); here we use only the most basic "flat" form of a CSP process, specified by a set of equations of the form below.

$$P = a_0 \rightarrow Q_0 \mid a_1 \rightarrow Q_1 \mid \ \ldots \ \mid a_{n-1} \rightarrow Q_{n-1}$$

The meaning is that process P evolves to process Q_0 on action a_0; to Q_1 on action a_1; and so forth. The actions need not be distinct: $P = a \rightarrow Q_0 \mid a \rightarrow Q_1$ represents a non-deterministic choice between Q_0 and Q_1 on action a. One may also view P as a state in a state machine, with the equation specifying the transitions at state P: the machine moves to state Q_0 on action a_0; to state Q_1 on action a_1; and so forth. The special process STOP has no outgoing transitions at all and thus represents a dead-end state. An entire state machine is thus described by a set of interrelated equations, one for each state.

In CSP, processes communicate only through an instantaneous, synchronized interaction on a common public action. If processes P and Q have a common action a, and if P may evolve to P' on a while Q may evolve to Q' on a, then their concurrent composition, denoted $P \parallel Q$, may *jointly* evolve to $P' \parallel Q'$ through synchronization on action a. On the other hand, private actions are unsynchronized; so that if P may evolve to P'' on a private action b, the composition $P \parallel Q$ may evolve to $P'' \parallel Q$ on b.

Coordination Problems. In the examples below, E denotes the single environment process with public actions $a_0, a_1, \ldots$ and private action b. The temporal specification of desired behavior, φ, is fixed to be "finitely many b actions"; this may be represented in temporal logic as $\Diamond\Box(\neg b)$. The goal is to construct a coordinator process, M, such that the combined system, $E \parallel M$, is (1) free of deadlock, and (2) satisfies φ on each of its infinite executions. If such a process exists, it is called a *solution* to the coordination problem (E, φ). A coordination problem is called *realizable* if it has a solution, and *unrealizable* otherwise. The examples illustrate both outcomes.

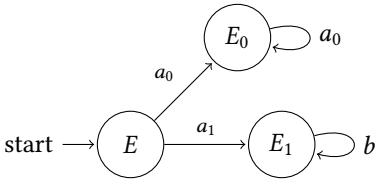

Fig. 2. Example 1: Environment E.

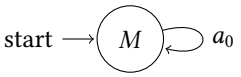

Fig. 3. A coordinator M.

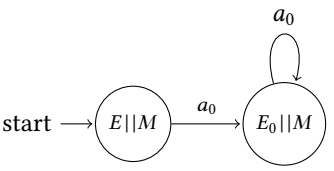

Fig. 4. $E||M$ satisfies $\Diamond\Box(\neg b)$.

Example 0: Non-blocking. The first example illustrates how a coordinator can guide the system so that it is deadlock-free. Consider E defined as follows.

$$E = a_0 \rightarrow E_0 \mid a_1 \rightarrow \mathsf{STOP}$$
$$E_0 = a_0 \rightarrow E_0$$

Let $M = a_0 \rightarrow M$. As E and M may synchronize only on a_0, the composition $E \parallel M$ is deadlock-free and has a single infinite computation $E \parallel M \xrightarrow{a_0} E_0 \parallel M \xrightarrow{a_0} E_0 \parallel M \ldots$, which trivially satisfies the specification. Hence M is a solution to this coordination problem.

Example 1: Realizability. Consider E defined as follows (Fig 2).

$$E = a_0 \rightarrow E_0 \mid a_1 \rightarrow E_1$$
$$E_0 = a_0 \rightarrow E_0$$
$$E_1 = b \rightarrow E_1$$

The self-loop of b-actions at E_1 violates the specification. Defining $M = a_0 \rightarrow M$ (Fig 3) ensures that the composed system $E \parallel M$ has a single joint computation, $E \parallel M \xrightarrow{a_0} E_0 \parallel M \xrightarrow{a_0} E_0 \parallel M \ldots$ (Fig 4), that entirely avoids the state E_1. As in Example 0, the structure of M guides the system away from an execution that would violate the specification.

Example 2: Unrealizability. Let E be defined as follows.

$$E = a_0 \rightarrow E_0 \mid a_0 \rightarrow E_1$$
$$E_0 = a_0 \rightarrow E_0$$
$$E_1 = b \rightarrow E_1$$

In this case, the coordination problem has no solution. Assume, to the contrary, that M is a solution. Then M must have a transition on a_0, otherwise the system $E \parallel M$ deadlocks. Denote the successor of this transition as M_0. But then $E \parallel M$ has the infinite joint computation $E \parallel M \xrightarrow{a_0} E_1 \parallel M_0 \xrightarrow{b} E_1 \parallel M_0 \xrightarrow{b} \ldots$ on which b occurs infinitely often, violating the specification.

Fairness for CSP Programs. A natural notion of fairness for a group of CSP processes, analogous to strong fairness, considers a computation to be *unfair* if there is a pair of processes that are ready to interact at infinitely many points along the computation, but do so only finitely many times. The assumption of fairness can make a synthesis problem easier to solve, as it limits the set of computations that must be examined for violations of the specification. Unfortunately, the joint computation of $E \parallel M$ constructed in Example 2 is fair, as the environment and coordinator can never interact once the environment is in state E_1, which only offers the private action b. So the coordination problem in Example 2 has no solution even under fairness.

Example 3: Realizability under fairness. Let E be defined as follows.

$$E = a_0 \rightarrow E_0 \mid a_0 \rightarrow E_1$$
$$E_0 = a_0 \rightarrow E_0$$
$$E_1 = b \rightarrow E_1 \mid a_0 \rightarrow E_0$$

Let $M = a_0 \rightarrow M$. Then $E \parallel M$ has a computation which violates the specification, where the environment enters state E_1 and then loops forever on the b action. That computation is unfair, however, as it is possible throughout for M and E_1 to synchronize on a_0, but such an interaction never occurs. All other computations of $E \parallel M$ satisfy the specification. So the synthesis problem is realizable under fairness, with M as a solution.

Example 4: Partial Information with a Non-deterministic Interface. Define E as follows.

$$E = a_0 \rightarrow E_0 \mid a_0 \rightarrow E_1$$
$$E_0 = a_0 \rightarrow E_0$$
$$E_1 = a_1 \rightarrow E_1$$

Any coordinator must have an initial synchronization on a_0, otherwise the system deadlocks. After performing a_0, the environment may be either in state E_0 or in state E_1. In the first state, only action a_0 is enabled, while in the second, only a_1 is enabled. Hence, a coordinator must be structured to synchronize on *either* of these two actions: if it offers to synchronize only on a_0, then if the environment actually enters state E_1, the system will deadlock; and vice-versa if it offers only a_1. Any synthesis algorithm must resolve such situations where the coordinator has only partial information of the environment state. In this case, the process $M = a_0 \rightarrow M_0$ where $M_0 = a_0 \rightarrow M_0 \mid a_1 \rightarrow M_0$ is a solution.

Example 5: Partial Information through Hidden Actions. Consider E defined as follows.

$$E = a_0 \rightarrow E_0$$
$$E_0 = b \rightarrow E$$

This problem is unrealizable. Assume, to the contrary, that M is a solution. To avoid deadlock, M must synchronize on action a_0. Let M_0 denote the successor state. Note that E synchronously evolves to E_0 on a_0. To avoid deadlock, it must again be possible to synchronize on a_0 from M_0. Proceeding in this manner, one constructs an infinite computation of $E \parallel M$ with infinitely many a_0 actions; but this computation must also have infinitely many b actions in-between successive a_0 actions, so it violates the specification. A synthesis algorithm must, therefore, carefully track the effect of hidden actions.

3 BACKGROUND

3.1 Communicating Sequential Processes (CSP)

CSP has a rich notation and extensive algebraic theory [Hoare 1985; Roscoe 1997]. As described previously, we use only the basic "flat" format, and thus define a *CSP process* as a tuple $P = (S, \iota, \Sigma, \Gamma, \delta)$, where S is a finite set of states, $\iota \in S$ is a special start state, Σ is a set of publicly visible actions of the process, and Γ is a set of privately visible actions of the process. The sets Σ and Γ are disjoint. The transition relation $\delta : S \times (\Sigma \cup \Gamma) \rightarrow 2^S$ maps each state and action to a set of successor states. A transition from state s on action a to state t exists if $t \in \delta(s, a)$; it is also written as (s, a, t) or sometimes as $s \xrightarrow{a} t$.

A process is *deterministic* if for all state s and all actions a, $|\delta(s,a)| \leq 1$, and *non-deterministic* otherwise. An action a is said to be *enabled* at state s if there is a transition on the action a from state s.

An *execution* π from state s_0 is an alternating sequence of states and actions $\pi = s_0, a_0, s_1, a_1, \ldots$, ending at a state if finite, such that $s_{i+1} \in \delta(s_i, a_i)$ for every i (where $i \geq 0$ and $i < n-1$ if there are n states on the sequence). The subsequence $a_0, a_1, \ldots$ of actions is the *trace* of the execution, denoted as $\text{trace}(\pi)$. A *computation* is an execution from the initial state. It is *maximal* if either it is infinite, or it is finite, and no transitions are enabled from the last state of the computation. A state s is *reachable* if there is a finite computation ending at s.

Process Composition and Interaction. Let P and Q be CSP processes. Let X be a subset of their common public actions, i.e., $X \subseteq (\Sigma_P \cap \Sigma_Q)$. The composition of P and Q relative to X, denoted $P \parallel_X Q$, is a CSP process, with state set $S_P \times S_Q$, initial state (ι_P, ι_Q), public actions $(\Sigma_P \cup \Sigma_Q) \setminus X$, private actions $(\Gamma_P \cup \Gamma_Q \cup X)$, and a transition relation defined by the following rules.

- (Synchronized) For an action a in X, there is a transition from (s,t) to (s',t') on a if (s,a,s') is a transition in P and (t,a,t') a transition in Q.
- (Unsynchronized) For an action b in Γ_P or in $\Sigma_P \setminus X$ (i.e., private, or unsynchronized public action), there is a transition from (s,t) to (s',t) on b if there is a transition (s,b,s') in P. A similar rule applies to Q.

The definition forces processes P and Q to synchronize on actions in X; for other actions, the processes may act independently. For simpler notation, we write $\parallel$ instead of $\parallel_X$ when X equals $\Sigma_P \cap \Sigma_Q$.

Fairness. There are several ways of defining fairness in CSP (cf. [Francez 1986]). We use the following notion, analogous to strong fairness. Let system S be defined as the parallel composition of a number of processes $S_1, S_2, \ldots, S_n$. A computation of S is *unfair* if there is a pair of processes S_i, S_j (for distinct i, j) such that some interaction between S_i and S_j is enabled at infinitely many points along the computation, but the computation contains only finitely many interactions between S_i and S_j. Intuitively, a computation is unfair if a system scheduler ignores a pairwise process interaction even if it is infinitely often possible.

3.2 Linear Temporal Logic

We formulate *Linear Temporal Logic (LTL)* [Pnueli 1977] over an alphabet Σ, using the syntax $\varphi ::= \text{True} \mid a \in \Sigma \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$. The temporal operators are X(Next) and U (Until). The LTL semantics is standard. For an infinite sequence π over Σ, the satisfaction relation, $\pi, i \models \varphi$ for a natural number i, is defined as follows:

- $\pi, i \models \text{True}$ holds for all i;
- $\pi, i \models a \in \Sigma$ if $\pi(i) = a$;
- $\pi, i \models \neg\varphi$ if $\pi, i \models \varphi$ does not hold;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ if $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \mathsf{X}\varphi$ if $\pi, i+1 \models \varphi$; and
- $\pi, i \models \varphi_1 \,\mathsf{U}\, \varphi_2$ if there is $j : j \geq i$ such that $\pi, j \models \varphi_2$ and for all $k \in [i, j)$, it is the case that $\pi, k \models \varphi_1$.

The Boolean constant False and Boolean operators are defined as usual. Other temporal operators are also defined in the standard way: $\Diamond\varphi$ ("Eventually φ") is defined as $\text{True}\,\mathsf{U}\,\varphi$; $\Box\varphi$ ("Always φ") is defined as $\neg\Diamond(\neg\varphi)$. For an LTL formula φ, let $\mathcal{L}(\varphi)$ denote the set of infinite sequences π over Σ such that $\pi, 0 \models \varphi$. In the standard formulation of LTL, the alphabet Σ is the powerset of a set of "atomic propositions".

3.3 Non-deterministic Büchi and Universal co-Büchi Automata

LTL formulas can be turned into equivalent *Büchi automata*, using standard constructions (e.g., [Babiak et al. 2012]). A *Büchi automaton* A is specified as a 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") 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 in A.

A *universal co-Büchi automaton* U is also specified as a tuple $(Q, Q_0, \Sigma, \delta, G)$. In this case, green states are also known as "rejecting" states. A *run* r of the automaton on an infinite word σ over Σ is defined as before. The difference arises in the definition of acceptance: run r is accepting if a green state appears on it *finitely many times*. The language of U, denoted $\mathcal{L}(U)$, is the set of words for which *all* runs are accepting in this sense.

The complement of the language of a Büchi automaton can be viewed as the language of a *universal co-Büchi* automaton with the identical structure. Thus, every LTL formula φ has an equivalent universal co-Büchi automaton.

3.4 Temporal Synthesis

In the standard (synchronous) formulation of temporal synthesis, the goal is to generate a deterministic reactive program M that transforms inputs from domain $\mathcal{I}$ to outputs from domain $\mathcal{O}$. Such a program can be represented as a function $f : \mathcal{I}^* \to \mathcal{O}$. The program M (or its functional form f) is viewed as a generator of an output sequence in response to an input sequence. For an infinite input sequence $x = x_0, x_1, \ldots$, the generated output sequence $y = y_0, y_1, \ldots$ is defined by: $y_i = f(x_0, \ldots, x_{i-1})$, for all $i \geq 0$. (In particular, y_0 is the value of f on the empty sequence.) The function f can also be viewed as a strategy for Alice in a turn-based two-player game where Bob chooses the input sequence, while Alice responds at the i-th step with the value y_i defined on the history of the play so far. Note that Alice plays first, specifying y_0.

Pnueli and Rosner [Pnueli and Rosner 1989a] represent f equivalently as a labeled infinite full-tree, a form that we also use to formulate a solution to the coordination synthesis problem. A tree over $\mathcal{I}$ is a prefix-closed subset of $\mathcal{I}^*$; a full-tree is the tree $\mathcal{I}^*$. Each finite string over $\mathcal{I}$ represents a node of the full-tree: the root is ϵ, and for $\sigma \in \mathcal{I}^*$ and $a \in \mathcal{I}$, the string $\sigma; a$ is the a-successor of the node σ. A labeling of the tree is a function $\mu : \mathcal{I}^* \to \mathcal{O}$ that maps each node of the tree to an output value. It is easy to see that the function f defines a labeling, and vice-versa.

The advantage of this view is that the set of deterministic reactive programs is isomorphic to the set of labeled full-trees. Thus, given an LTL specification φ over the alphabet $(\mathcal{I} \times \mathcal{O})$, Pnueli and Rosner solve the LTL synthesis question (following Rabin [Rabin 1969]) by constructing a tree automaton that accepts only those labeled trees that represent programs that are solutions, and checking whether this automaton has a non-empty language. A labeled tree represents a solution if it is a full tree and if for every input sequence $x = x_0, x_1, \ldots$, the labels $y = y_0, y_1, \ldots$ on the path in the tree defined by x are such that the sequence of pairs $(x_0, y_0), (x_1, y_1), \ldots$ satisfies φ.

The formulation of M (or f) here corresponds to a Moore machine; there is an analogous formulation for Mealy machines. A specification is said to be *realizable* if there is a program that satisfies it; it is *unrealizable* otherwise.

3.5 Solution Methods for Temporal Synthesis

The tree-automaton view of the synthesis question, though mathematically elegant, is not a viable approach in practice, as the construction of a tree automaton involves complex constructions for determinization and complementation of automata on infinite words.

In practice, tools such as BoSy [Faymonville et al. 2017a] and Acacia+ [Bohy et al. 2012] adopt a different strategy called *bounded synthesis* [Filiot et al. 2009; Schewe and Finkbeiner 2007]. The essential idea is to pick a bound, N, on either the number of states in a solution, or the number of visits to a rejecting state in a universal co-Büchi automaton. The tool Acacia+ uses this bound to formulate the problem as a *safety* game, and searches for a winning strategy for Alice. In BoSy, the search is for a transition relation for M that has at most N states.

We summarize the BoSy search here, as we use BoSy as the back-end synthesis tool in our implementation. Given a universal co-Büchi automaton representing the LTL specification φ over alphabet $\mathcal{I} \times \mathcal{O}$ and state space Q, and a bound N on the number of states of M, BoSy searches for a deterministic Moore machine with state space $S = \{0 \ldots (N-1)\}$. The search is for (1) a transition function $T : S \times \mathcal{I} \to S$, (2) an output function $O : S \to \mathcal{O}$, (3) an inductive invariant $\theta \subseteq Q \times S$, and (4) a rank function $\rho : Q \times S \to [K]$, where K is a derived rank bound, such that these objects satisfy the standard deductive verification proof rule for universal automata [Manna and Pnueli 1987].

As described in [Faymonville et al. 2017b], these constraints can be encoded and solved in various ways, either as a propositional satisfiability (SAT) problem, or as a quantified boolean constraint (QBF) problem. The bounded synthesis approach is complete, as there is a known (worst-case exponential) limit on the size of a model for a specification automaton.

4 PROBLEM FORMULATION

We gather the environment agents $E_1, \ldots, E_n$ together as the process E. The set of public actions of E is denoted Σ, while the set of its private actions is denoted Γ.

Specifications. The maximal computations of a CSP process may either be finite, ending in a state with no successor, or are infinite. The semantics of LTL is defined only over infinite sequences of actions, but a correctness specification should accommodate both types. Thus, we define a specification φ as a pair (φ_S, φ_L), where φ_S is a set of finite sequences over $\Sigma \cup \Gamma$ and φ_L is a set of infinite sequences over $\Sigma \cup \Gamma$. To formulate a synthesis algorithm, we assume that the *complement* sets of φ_S and φ_L are definable by finite automata over $\Sigma \cup \Gamma$. For φ_S, this is an automaton over finite words; for φ_L, this is a Büchi automaton over infinite words.

As introduced in Section 2, we also consider fairness. An infinite computation π of $E \parallel M$ is *unfair* if there are infinitely many points on π where an interaction between the two processes is enabled, but only finitely many actual interactions on π. A maximal fair computation of $E \parallel M$ satisfies φ in one of two ways, based on its projection over $(\Sigma \cup \Gamma)$: either the projection is finite and belongs to φ_S, or it is infinite and belongs to φ_L. If M has private actions, it is possible for an infinite computation to have a finite projection over $(\Sigma \cup \Gamma)$.

Coordination Synthesis. The synthesis problem has been introduced informally; it is defined precisely below.

Definition 4.1 (Coordination Synthesis Problem). Given an environment process $E = (S, \iota, \Sigma, \Gamma, \delta)$ and a specification φ over $(\Sigma \cup \Gamma)$, construct a process M (if one exists) with public action set Σ such that all maximal fair computations of $E \parallel_\Sigma M$ satisfy φ.

A problem instance (E, φ) is *realizable* if there is a process M meeting the requirements stated above; it is *unrealizable* otherwise. A process M is *non-blocking* for process E if all maximal

computations of $E \parallel M$ are infinite; i.e., the composition is free of deadlock. By setting φ_S to the empty set, every solution to the problem instance (E, φ) is non-blocking, as no maximal finite trace can satisfy $\varphi_S = \emptyset$.

Restricting the solution space. This problem formulation allows a solution M to be non-deterministic and have its own set of private actions, distinct from $\Sigma \cup \Gamma$. (The specification cannot, of course, reference those actions.) A CSP process can exhibit two types of non-determinism: *internal nondeterminism*, where a state has more than one possible successor on a private action, and *external nondeterminism*, where a state has more than one possible successor on a public action. Intuitively, the first defines a choice that is always enabled and is to be resolved by the process itself, while the second is a choice that is resolved only through synchronization with an external process.

The theorem below shows that there is no loss of generality in restricting the search to deterministic processes without private actions. This is a simple but important observation, as the behavior of a deterministic process can be viewed as a tree, and analysis of such trees is the basis of the method presented in the following sections.

Theorem 4.2. *A synthesis instance (E, φ) is realizable if, and only if, it has a deterministic solution process that has no private actions.*

Proof. Let M be a process that is a solution to the instance (E, φ). Let $\Delta = \Sigma \cup \Gamma$. The proof proceeds in two stages. First, private actions of M are hidden to produce M'; as those actions cannot be referenced in φ, the process M' is also a solution. Next, non-determinism in public actions of M' is eliminated to obtain M'' that is also a solution. By construction, M'' is deterministic and has no private actions.

In the first stage, we eliminate private transitions in M (this set is necessarily disjoint from Δ) to obtain M' that is also a solution The states of M' are those of M. The transition relation of M' is defined as follows: for states s, t and public action a, a triple (s, a, t) is in the transition relation of M' if, and only if, there is a path with the trace $\beta; a; \beta'$ from s to t, where β and β' are sequences of private actions of M. Consider any maximal fair computation x of $E \parallel M'$. This computation can be turned into a maximal computation y of $E \parallel M$ simply by replacing each transition of M' by the sequence of transitions of M used to define it. (A subtle point is that it is possible for x to be finite and the corresponding maximal sequence y to be infinite if there is an infinite path (a "tail") of private transitions of M originating at the final state of x.) The two computations have identical traces when projected on Δ, and if x is infinite and fair, so is y. As y satisfies φ by the assumption that M is a solution, so does x.

In the second stage, we eliminate external nondeterminism from M' to obtain M'' that is also a solution but has no external non-determinism. This could be done by determinizing M' using the standard subset construction, but there is a simpler construction that does not incur the worst-case exponential blowup; it simply restricts the transition relation. M'' has the same states and initial state as M'; however, its transition relation, T'', is such that $T''(s, a) \subseteq T'(s, a)$, and $|T''(s, a)| = 1$ iff $|T'(s, a)| \geq 1$. Informally, T'' chooses one of the successors of T' on action a from each state. By construction, M'' is externally deterministic and has no private actions.

Unlike with the subset construction, the traces of M'' obtained by restriction could be a proper subset of those of M', but it is still a solution. Consider any maximal fair computation x of $E \parallel M''$. By construction, x is also a maximal computation of $E \parallel M'$ and is fair as the construction preserves enabledness. As M' is a solution, x satisfies φ. (A subtle point is that if x ends in a dead-end state of M'', this state is also a dead-end state of M', as the restriction only removes duplicate transitions.) □

5 TREE VIEWS AND AUTOMATON-THEORETIC SYNTHESIS

This section formulates the automaton-based synthesis procedure. As described in Section 3.4, the goal is to formulate conditions under which a labeled fulltree represents a solution to the synthesis question. However, one cannot simply reuse the constructions and results of Pnueli and Rosner, as the coordination synthesis question has a distinct formulation from the input-output form they consider. The theorems in this section pin down the conditions for validity and show how to represent them as automata. Detailed proofs and constructions are in [Bansal et al. 2019].

In a nutshell, the technical development proceeds as follows.

(1) By Theorem 4.2, it suffices to limit a solution M to be externally deterministic and have no private actions. The semantics of such a process can be represented as an infinite labeled fulltree, where each node is labeled with a *set* of public actions and each edge with a public action.
(2) We show that a fulltree labled in this manner is *invalid* –i.e., it does not represent a solution – if and only if it has an infinite path satisfying a linear-time property derived from the specification φ and the environment process E.
(3) We show that this failure property may be represented by a non-deterministic Büchi automaton, $\mathcal{B}$, constructed from E and a Büchi automaton, $\mathcal{A}$, for the negation of φ. Partial knowledge and asynchrony are both handled in this construction.
(4) The *valid* fulltrees are, therefore, those where every path in the tree is accepted by the complement of $\mathcal{B}$, which is represented by the same structure by viewing it as a *universal* co-Büchi automaton. Automaton $\mathcal{B}$ has a number of states that is *linear* in the number of states of E and of $\mathcal{A}$. Its transition relation, however, is of size *exponential* in the number of public interface actions.
(5) We give a fully symbolic construction of $\mathcal{B}$ (in Section 6) to ameliorate the exponential blowup in the size of its transition relation. The symbolic $\mathcal{B}$ is in a form that can be solved by a number of temporal synthesis tools. The constructed solution (if one exists) is in the form of a Moore machine. We show how to transform this back to a CSP process representing the coordinator M.
(6) We show that the synthesis problem is PSPACE-hard in the size of the environment for a fixed specification. It is well known that the synthesis problem is 2-EXPTIME-complete in the size of the LTL specification [Pnueli and Rosner 1989a].

The first few steps explain the issues in terms of fulltrees, which is easier to follow. The final synthesis procedure is *not* based on tree-automaton constructions; instead, it utilizes the co-Büchi word automaton $\mathcal{B}$, a format supported by current tools.

5.1 Labeled Fulltrees and Deterministic CSP Processes

A *tree* t over alphabet Σ is a prefix-closed subset of Σ^*. A *node* of tree t is an element of Σ^*. For a set $\mathcal{L}$ of *labels*, an $\mathcal{L}$-labeled tree is a pair (t, μ) where $\mu : t \to \mathcal{L}$. I.e., μ assigns to each node of the tree an element of $\mathcal{L}$, its label. The *full* tree over Σ is the set Σ^*. For a deterministic CSP process M, one can generate a full Σ-tree labeled with 2^Σ as follows. The labeling function, which we denote μ_M, assigns to tree node σ the label defined as follows. By determinism, there is at most one computation of M with trace σ. If this computation exists, let the label $\mu_M(\sigma)$ be the set of actions that are enabled at the state at the end of the computation. Otherwise, let $\mu_M(\sigma)$ be the empty set. We refer to this labeled tree as $\mathsf{fulltree}(M)$.

Conversely, given a full Σ-tree (t, μ) labeled with 2^Σ, one can extract a deterministic (infinite-state) CSP process, $P = \mathsf{proc}(t, \mu)$, as follows. The state space of P is the set of tree nodes. The

initial state is the node ϵ. There is a transition from state σ to state δ on action a iff $a \in \mu(\sigma)$ and $\delta = \sigma; a$. We then have the following theorem.

Theorem 5.1. *For a deterministic CSP process M, the process $M' = \mathsf{proc}(\mathsf{fulltree}(M))$ is bisimular to M.*

As bisimular processes behave identically in the context E, we obtain

Corollary 5.2. *The coordination synthesis instance (E, φ) is realizable iff there is a full labeled Σ-tree (t, μ) labeled by 2^Σ such that $\mathsf{proc}(t, \mu)$ is a solution to (E, φ).*

5.2 Recognizing Valid Fulltrees

We focus on recognizing the kinds of fulltrees defined by Corollary 5.2. It is actually easier to formulate properties of a fulltree t that *exclude* $\mathsf{proc}(t, \mu)$ from being a solution. From now on, we fix a "labeled tree" to mean a Σ-fulltree labeled with 2^Σ.

A *path* in a Σ-tree is a word (finite or infinite) over Σ. A path $\pi = a_0; a_1; \ldots$ in a labeled tree induces a *labeled path*, which is an alternating sequence $L_0; a_0; L_1; a_1; \ldots$ where for each i, $L_i = \mu(a_0, \ldots, a_{i-1})$ is the label of the node $(a_0, \ldots, a_{i-1})$. If the path π is finite, the labeled path ends with the label $\mu(\pi)$, which we refer to as $\mathsf{endL}(\pi)$. Path π is said to be *consistent* iff $a_i \in L_i$ for all i, for the labeled path induced by π. Every consistent path π in tree (t, μ) is in correspondence with a computation of $\mathsf{proc}(t, \mu)$. As a computation is a process in itself, we abuse notation slightly and refer to this computation as the process $\mathsf{proc}(\pi)$.

Definition 5.3 (Full-tree violation). We say that a labeled fulltree (t, μ) violates (E, φ) if one of the following holds.

(A) There is a consistent *finite* path π in the tree such that some finite computation γ in $E \parallel \mathsf{proc}(\pi)$ is maximal, $\mathsf{trace}(\gamma)$ has π as a subword, and $\mathsf{trace}(\gamma)$ is not in φ_S, or
(B) There is a consistent *finite* path π in the tree such that for some infinite computation γ of $E \parallel \mathsf{proc}(\pi)$, $\mathsf{trace}(\gamma)$ has π as a subword, $\mathsf{trace}(\gamma)$ is not in φ_L, and no action from $\mathsf{endL}(\pi)$ is enabled from some point on in γ, or
(C) There is a consistent *infinite* path π in the tree such that for some infinite computation γ of $E \parallel \mathsf{proc}(\pi)$, $\mathsf{trace}(\gamma)$ has π as a subword and is not in φ_L.

Illustrative Example. Consider the CSP environment E defined as follows.

$$E = a_0 \rightarrow STOP \mid a_0 \rightarrow E_0 \mid b_0 \rightarrow E_1$$
$$E_0 = b_1 \rightarrow E_0$$
$$E_1 = a_1 \rightarrow E_1$$

The public actions are a_0 and a_1, and the private actions are b_0 and b_1. Let φ_S be the empty set and φ_L be the set of sequences with infinitely many a_1. Figure 5 depicts three labeled trees. Tree (a) has a consistent finite path $\pi = a_0$. This path satisfies condition (A), as $E \parallel \mathsf{proc}(\pi)$ deadlocks. Tree (b) has a consistent finite path $\pi = a_0$. This satisfies condition (B) as $E \parallel \mathsf{proc}(\pi)$ has an infinite computation with trace $a_0(b_1)^\omega$, which is not in φ_L. Tree (c) represents $\mathsf{fulltree}(M)$ for process $M = a_1 \rightarrow M$. The only consistent path is $(a_1)^\omega$, which induces a computation in $E \parallel M$ with trace $b_0(a_1)^\omega$, which satisfies φ_L. Therefore tree (c) is valid for (E, φ).

Theorem 5.4. *Fix a synthesis instance (E, φ). For any labeled fulltree (t, μ), the process $\mathsf{proc}(t, \mu)$ is a solution if, and only if, (t, μ) does not violate (E, φ).*

Proof. We will show the contrapositive.

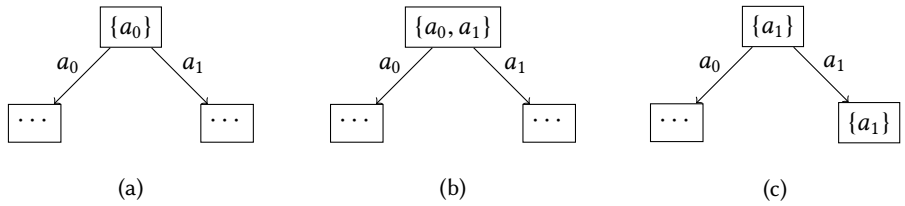

Fig. 5. Candidate trees (showing only first two levels). Nodes are labeled with sets of actions; $\cdots$ indicates a don't-care label.

(left-to-right) Suppose that (t, μ) violates (E, φ). Then one of (A)-(C) holds. We show that process $M = \mathsf{proc}(t, \mu)$ cannot be a solution.

Suppose case (A) holds. The maximal finite computation γ of $E \parallel \mathsf{proc}(\pi)$ is also a maximal computation of $E \parallel M$, as π corresponds to a computation of M. As $\mathsf{trace}(\gamma)$ is not in φ_S, M is not a solution.

Suppose case (B) holds. The infinite computation γ of $E \parallel \mathsf{proc}(\pi)$ is also a computation of $E \parallel M$, which includes all the actions of π. As π is finite, γ has only finitely many synchronization actions. Moreover, none of the actions of $\mathsf{endL}(\pi)$ are enabled on γ from some point on. Thus, no synchronization between M and E is enabled on γ from that point on, so γ is fair, but its trace is not in φ_L.

Suppose case (C) holds. The computation γ of $E \parallel \mathsf{proc}(\pi)$ is an infinite computation of $E \parallel M$, which is fair as it has infinitely many synchronizations, as $\mathsf{trace}(\gamma)$ includes π. However, $\mathsf{trace}(\gamma)$ is not in φ_L.

(right-to-left) Suppose $M = \mathsf{proc}(t, \mu)$ is not a solution for (E, φ). We show that (t, μ) must violate (E, φ). As $E \parallel M$ fails to satisfy φ, there are two possibilities.

The first possibility is that of a maximal finite computation of $E \parallel M$ whose trace is not in φ_S. This computation corresponds to a finite consistent path π in the tree (t, μ) that meets condition (A).

The other possibility is that there is an infinite fair computation γ of $E \parallel M$ whose trace is not in φ_L. There are two types of fair computations.

In the first type, γ has only finitely many synchronization actions followed by an infinite suffix where no synchronization is enabled with M. As M has no private actions, the suffix of γ consists solely of private actions of E. The shortest finite prefix of γ containing all synchronization actions defines a consistent finite path π in (t, μ). The set of enabled actions of M at the end of this prefix of γ is $\mathsf{endL}(\pi)$. Thus, π satisfies the requirements of condition (B).

In the second type, γ has infinitely many synchronization actions. This induces an infinite computation of M, and a corresponding consistent infinite path π of (t, μ) meeting condition (C). □

5.3 Automata-Theoretic Synthesis

We now describe how to construct Büchi automata that recognize the consistency of a labeled path, and whether properties (A)-(C) hold. The individual automata recognize infinite sequences over an input alphabet $\Sigma \times 2^{\Sigma}$. An input sequence of the form $(a_0, L_0), (a_1, L_1), \ldots$ represents the labeled path $L_0; a_0; L_1; a_1; \ldots$. Note that Γ actions do not appear in the input alphabet, they are eliminated in the process of constructing the automaton. The alphabet, and hence the size of the automaton transition relation is exponential in $|\Sigma|$.

Checking for consistency. The first automaton accepts an input sequence of the form above if it corresponds to a consistent path, i.e., if $a_i \in L_i$ for all i. This automaton has a constant number of states.

Checking condition (A). Let $\mathcal{A}_S$ be a finite automaton for the negation of φ_S. The automaton for (A) is a product of $\mathcal{A}_S$ and E. It guesses a *finite* computation γ of E, where the sub-sequence of public actions on trace(γ) is $a_0, a_1, \ldots, a_{n-1}$ (for $n \geq 0$). The automaton also concurrently guesses a run of the finite-word automaton $\mathcal{A}_S$ on trace(γ), which also includes private actions of E. It accepts if this automaton is in an accepting state after the last symbol, the final state e of γ has no transitions on private actions, and none of the public transitions enabled at e are in L_n, the final label. This automaton has $|\mathcal{A}_S| * |E|$ states.

Checking condition (B). Let $\mathcal{A}_L$ be a finite Büchi automaton for the negation of φ_L. The automaton for (B) guesses, as in the construction for case (A), a finite computation γ of E where the sub-sequence of public actions on trace(γ) is $a_0, a_1, \ldots, a_{n-1}$ (for $n \geq 0$). Concurrently, it guesses a run of the infinite word automaton $\mathcal{A}_L$ on trace(γ) to an automaton state, say, q. It then guesses a further infinite execution γ' of E, starting at the final state e of γ, such that all transitions on γ' are on private actions of E, and it checks that at each intermediate state of γ', none of the enabled public transitions are in the final label L_n. It concurrently extends the run of $\mathcal{A}_L$ on trace(γ') from state q, and accepts if this run is accepting for $\mathcal{A}_L$. This automaton has $|\mathcal{A}_L| * |E|$ states.

Checking condition (C). Let $\mathcal{A}_L$ be a finite Büchi automaton for the negation of φ_L. The automaton for (C) guesses an infinite computation γ of E such that the sequence of public actions on trace(γ) is $a_0, a_1, \ldots$. It concurrently simulates a run of the automaton $\mathcal{A}_L$ on trace(γ), accepting if this run accepts. This automaton has $|\mathcal{A}_L| * |E|$ states.

Constructing the transformed specification automaton. We construct an automaton by taking the union of the automata for (A)-(C) and intersecting the union with the automaton that checks consistency of the input sequence. The result is a non-deterministic Büchi automaton which accepts an input sequence if it is consistent and satisfies one of (A)-(C). The same structure can be viewed as a *universal* co-Büchi automaton that accepts a path of a labeled full tree either if it is inconsistent, or if it does not satisfy one of (A)-(C). This is the *transformed specification*, represented as an automaton $\mathcal{B}$, which has $(|\mathcal{A}_S| + |\mathcal{A}_L|) * |E|$ states. In the classical setting (cf. [Pnueli and Rosner 1989a]), this universal automaton is converted to a tree automaton which checks whether a labeled tree is a valid fulltree. The synthesis problem is realizable iff the language of this tree automaton is non-empty. This construction is mathematically elegant but impractical due to the need for complex determinization constructions. In the alternative setting of bounded synthesis [Finkbeiner and Schewe 2013], the universal co-Büchi automaton suffices and is used directly in the check for realizability.

5.4 Hardness of Coordination Synthesis

We show that the coordination synthesis problem is PSPACE-hard in terms of the size of the environment, E, keeping the specification formula fixed.

Theorem 5.5. *Coordination Synthesis is PSPACE-hard in $|E|$ for a fixed specification.*

Proof. The proof is by reduction from a standard PSPACE-complete problem: given a finite automaton, A, determine whether the language of A is *not* the universal language over its alphabet, Σ. Without loss of generality, we suppose that A is complete – i.e., each state has a transition on every letter in Σ.

From A, we construct a synthesis instance (E, φ) as follows. The environment E is a copy of A, with two additional states, accept and reject, and three additional letters, $\sharp$, $+$, and $-$. Its transition relation is that of A, together with the following additional transitions. First, from each accepting state of A, there is a transition on $\sharp$ to the accept state. The accept state has a self-loop transition on $+$. Second, from each non-accepting state of A, there is a transition on $\sharp$ to the reject state. The reject state has a self-loop transition on letter $-$. Note that E has no private actions but, in general, will have non-deterministic choice on public actions.

The specification φ is given by $\varphi_S = \emptyset$ (i.e., non-blocking solutions are required) and φ_L being the LTL formula $\Diamond(\sharp \wedge \mathsf{X}\Box(-)))$. We show that the language of A is non-universal if, and only if, the synthesis problem (E, φ) has a solution.

In the left-to-right direction, suppose that A has a non-universal language. Let w be a word that is not accepted by A. Let M be the deterministic process with a single execution with trace $w\sharp(-)^\omega$. We claim that M is a solution for (E, φ). As A is complete, and as every run of w on A ends in a non-accepting state, every maximal computation of $E \parallel M$ is infinite and has the trace $w\sharp(-)^\omega$, which meets the specification. Hence, M is a solution.

In the other direction, suppose that A has a universal language but there is a solution M. Then $E \parallel M$ must have at least one infinite computation, say σ, and $\mathrm{trace}(\sigma)$ must satisfy φ_L. Thus, this trace has form $w\sharp(-)^\omega$ for some w. Let s be the state of M on σ after the transition sequence w. As w must be in the language of A, there is a run of A on w to some accepting state, say t. Then there is a computation of $E \parallel M$ with trace w to the joint state (s, t). As $(s, \sharp, s')$ is a transition on σ for some s', this computation can be extended to the joint state (s', accept). To avoid deadlock, there must be an infinite exection from s' with trace $(+)^\omega$. But then there is an infinite computation of $E \parallel M$ with trace $w(+)^\omega$, which does not satisfy φ_L, a contradiction. □

6 SYMBOLIC CONSTRUCTIONS

In this section, we give a fully symbolic construction for the automaton of the transformed specification. The symbolic form ameliorates the exponential blowup in the size of its transition relation. This construction forms the basis of the implementation described in Section 7. Rather than construct separate automata for conditions (A)-(C), the construction builds a single automaton.

- The states are either (1) special states Fail and Sink, or (2) normal states of the form (q, r, e), where q is a state of $\mathcal{A}_L$, r is a state of $\mathcal{A}_S$, and e is an environment state.
- The initial state is (q_0, r_0, e_0) where all components are initial in their respective structures.
- The input symbols have the form (a, L, g), where a is an public action, L is a set of public actions, and g is a Boolean marking the transition as being "green" (accepting) if true.
- The transition relation of the joint automaton is described as a predicate, T(current_state, input_symbol, next_state). On all input symbols, the special states Fail and Sink have a self loop; i.e., the next state is the same as the current state. The transition relation for normal states is defined below.
- The green (accepting) edges of the automaton are those transitions where g is true.
- The green (accepting) states of the automaton are the special state Fail and every state (q, r, e) where q is a green state of $\mathcal{A}_L$.

For a normal state (q, r, e) and input symbol (a, L, g), the transition relation T is defined as follows. It relies on several auxiliary relations, each relates to the check shown in parenthesis.

(1) If $\mathsf{Efail}(r, e, L)$ holds, the successor state is Fail (Condition (A)).
(2) Otherwise, if $\mathsf{noSynch}(q, r, e, L)$ holds, the successor state is Fail (Condition (B)).
(3) Otherwise, if $\mathsf{Esink}(a, e, L)$ holds, the successor state is Sink (Consistency).

(4) Otherwise, if $\mathsf{normalTrans}((q, r, e), (a, g), (q', r', e'))$ holds, the successor is the normal state (q', r', e') (Condition (C)) .

The auxiliary relations represent the existence of paths, and are defined in the form of a least fixpoint. Each of these fixpoint computations, as well as the construction of the transition relation, can be computed symbolically using BDDs. We provide the fixpoint formulations below. In these formulations, for an action c (public or private), the joint transition $\mathsf{Joint}((q, r, e), c, (q', r', e'))$ holds iff $\mathcal{A}_L(q, c, q')$, $\mathcal{A}_S(r, c, r')$, and $E(e, c, e')$ all hold.

Following condition (A), predicate $\mathsf{Efail}(r, e, L)$ holds if there is a path consisting only of private transitions from state e in E to an end-state e' that fails the φ_S property. The state e' can have no private transitions and no enabled public actions in L. Being a reachability property, the predicate is defined as the least fixpoint of $Y(r, e, L)$ where

- (Base case) If r is a final state of $\mathcal{A}_S$, e has no private transitions, and none of its enabled public actions is in L, then $Y(r, e, L)$ is true, and
- (Induction) If there exists r', e' and a private action b such that $Y(r', e', L)$, $\mathcal{A}_S(r, b, r')$ and $E(e, b, e')$ hold, then $Y(r, e, L)$ holds.

Following condition (B), predicate $\mathsf{noSynch}(q, r, e, L)$ holds if there exists an infinite path consisting of only private transitions that fails the φ_L property, and from some point on L does not synchronize with the public actions enabled on the path. One can easily argue that it is sufficient to find an *lasso*, where the stem consists of only private transitions while no state in the lasso loop synchronizes with L. Formally, $\mathsf{noSynch}(q, r, e, L)$ holds if there exists a private action b, states q_0, r_0, e_0, g_0 and q_1, r_1, e_1, g_1 such that all of the the following conditions hold

- $\mathsf{Eprivate}((q, r, e), g_0, (q_0, r_0, e_0))$ (the stem),
- $\mathsf{Joint}((q_0, r_0, e_0), b, (q_1, r_1, e_1))$ (the link),
- $\mathsf{GenEprivate}((q_1, r_1, e_1), g_1, L, (q_0, r_0, e_0))$ (the loop), and
- $g \equiv g_1$.

The predicate $\mathsf{Eprivate}((q, r, e), g, (q', r', e'))$ holds if there is a path consisting only of private transitions from state e in E to e', a run of the $\mathcal{A}_S$ automaton on this path from state r to r', a run of the $\mathcal{A}_L$ automaton on the same path from state q to q', and g is true if one of the states on the $\mathcal{A}_L$ automaton run is a green (i.e., Büchi accepting) state. Being a reachability property, this can be defined as the least fixpoint of $Z((q, r, e), g, (q', r', e'))$, where

- (Base case) If $q = q', r = r', e = e'$ then $Z((q, r, e), g, (q', r', e'))$ holds, and g is true iff q is green, and
- (Induction) If $Z((q, r, e), g_0, (q_0, r_0, e_0))$ and $J((q_0, r_0, e_0), b, (q', r', e'))$ for a private action b, then $Z((q, r, e), g, (q', r', e'))$ holds, with g being true if g_0 is true or q' is green.

Predicate $\mathsf{GenEprivate}((q, r, e), g, L, (q', r', e'))$ generalizes $\mathsf{Eprivate}((q, r, e), g, (q', r', e'))$ by additionally ensuring that no state along the run from e to e' in E enables a public action in L.

Predicate $\mathsf{Esink}(a, e, L)$ holds if either $a \notin L$ (inconsistent), or $\neg\mathsf{enabled}(a, e)$ (blocking) where predicate $\mathsf{enabled}(a, e)$ holds if there is a path in E consisting only of private transitions from state e to a state from which there is a transition on public symbol a. Being a reachability property, this predicate is defined as the least fixpoint of $X(a, e)$ where

- (Base case) If e has a transition on a, then $X(a, e)$ holds.
- (Induction) If there exists e' and a private action b such that $E(e, b, e')$ and $X(a, e')$ hold, then $X(a, e)$ holds.

Following condition (C), predicate $\mathsf{normalTrans}((q, r, e), (a, g), (q', r', e'))$ holds if there is a path of private actions to the normal target state (q', r', e') with a single public transition on a, along

with matching runs of the two automata. Formally, normalTrans$((q, r, e), (a, g), (q', r', e'))$ holds if there exists states q_0, r_0, e_0, g_0 and q_1, r_1, e_1, g_1 such that all of the the following conditions hold

- Eprivate$((q, r, e), g_0, (q_0, r_0, e_0))$,
- Joint$((q_0, r_0, e_0), a, (q_1, r_1, e_1))$,
- Eprivate$((q_1, r_1, e_1), g_1, (q', r', e'))$, and
- $g \equiv g_0 \vee g_1$.

Encoding into Pnueli-Rosner synthesis. The resulting automaton has transitions on (a, L, g). The green transitions can be converted to green states by a simple construction (omitted) that adds a copy of the state space. This automaton operates on input sequences of the form $(a_0, L_0), (a_1, L_1), \ldots$. That matches the form of the *synchronous* Pnueli-Rosner model where automata operate on input sequences of the form $(x_0, y_0), (x_1, y_1), \ldots$, where the x's represent input values and the y's the output value chosen by the synthesized process. By reinterpreting the a's as x's and L's as y's, one can use existing tools for synchronous synthesis to check for realizability and produce a solution if realizable. In this re-interpretation, if the public alphabet has n symbols, there are $\log(n)$ bits for x (i.e., a), and n bits for y (i.e., L).

Mapping a solution back to CSP. The synchronous synthesis tools produce a deterministic Moore machine as a solution to the reinterpreted synthesis problem. Such a machine maps a sequence of x-values (i.e., a sequence of public actions, reinterpreted) to a y-value (i.e., a set of public actions, reinterpreted). That is precisely the CSP process M, defined as follows. The states of M are the states of the Moore machine. There is a transition (s, a, t) in M if, and only if, a is included in the output label y of the Moore machine at state s, and the machine has a transition from s to t on input $x = a$.

7 IMPLEMENTATION AND CASE STUDIES

7.1 Implementation and Workflow

We implement the coordination synthesis algorithm in about 1500 lines of Python code. The implementation uses the FDR4 CSP tool [Thomas Gibson-Robinson 2014][1] to read in a CSP description of the environment written in the language CSP_M, and produce a flattened description in the form of a state machine for the environment E. It uses features of the SPOT system [Duret-Lutz et al. 2016][2] to read in LTL formulas that define the specifications φ_S and φ_L, and to convert those to the appropriate automaton form. The core of the construction (about 1000 lines of Python) implements the symbolic specification automaton construction described in Sections 5 and 6. The resulting automaton, $\mathcal{B}$, is simplified using the SPOT tool autfilt to, for instance, remove dead states. It is then supplied to the synchronous synthesis engine BoSy [Faymonville et al. 2017b][3]. BoSy produces a Moore machine, which is converted into a CSP process as described at the end of Section 6. We are grateful to the authors of these and supporting tools for making the tools freely available.

As defined in Section 5, the automaton $\mathcal{B}$ operates on inputs from the alphabet $\Sigma \times 2^{\Sigma}$, where Σ is the set of public interface actions of the environment E. Thus, an input symbol has the form (a, L), where $a \in \Sigma$ and L is a subset of Σ. BoSy expects these values to be encoded as bit vectors defining the input, denoted x, and output, denoted y, of the synchronous synthesis problem. For simplicity, we choose a 1-hot encoding for the symbol a; i.e., we have a bit x_m for each symbol m in Σ, and ensure that exactly one of those bits is true – to represent symbol a, the bit x_a is set to

[1] FDR4 available at https://www.cs.ox.ac.uk/projects/fdr/
[2] SPOT available at https://spot.lrde.epita.fr/
[3] BoSy available at https://www.react.uni-saarland.de/tools/bosy/

true while all others are set to false. To encode the set L, we define bits y_m for each symbol m in Σ. Thus, a set L is encoded by setting y_m to be true for all m in L, and y_m to false for all m not in L.

Recall from Section 3 that BoSy encodes the synchronous synthesis problem as constraint solving, either propositional (with SAT solvers) or quantified (with QBF solvers). The QBF form has the quantifier prefix $\exists\forall\exists$, where the universal quantification is over the input variables; in effect, the SAT encoding is obtained by replacing the $\forall$ quantifier with a conjunction over all values of the input variables. In our case, the input variables represent elements of Σ, as explained above. As this set is not very large (for our examples), we can replace the $\forall$ quantifier with a conjunction over all elements of Σ, and obtain an equivalent SAT problem. Doing this in practice required a small modification to BoSy, as its default is to expand over *all* assignments to input variables, which would lead to a $|2^{\Sigma}|$ blowup in the expansion for our 1-hot encoding. Instead, we modify BoSy to expand over only 1-hot assignments to the input variables, which gives the expected linear $|\Sigma|$ blowup. We refer to our SAT encoding by OneHot-SAT.

In prior work on BoSy [Faymonville et al. 2017b] the authors report that QBF outperforms SAT. Much to our surprise, we observed the reverse: the SAT encoding handily outperforms QBF, as shown in the experiment tables later in this section. We use the state-of-the-art SAT solver CryptominiSAT [Soos et al. 2009].

All experiments were run on 8 CPU cores at 2.4GHz, 16GB RAM, running 64-bit Linux.

7.2 Case Studies

The objective of our case studies is to demonstrate the utility of coordination synthesis in designing coordination programs under various aspects of asynchrony, partial information and concurrency. We begin by testing our implementation on examples from Section 2 (Section 7.2.1), and then follow it up by designing a coordination program for a *smart thermostat* (Section 7.2.2) and an *arbiter* for concurrent processes (Section 7.2.3). We explore different aspects of the applicability of coordination synthesis in each of these studies, ranging from modeling choices to fairness in concurrency. Finally, we discuss the scalability challenges in Section 7.2.4.

7.2.1 Illustrative Examples from § 2. We begin with demonstrating the correctness of our implementation by synthesizing all specifications from the illustrative examples given in Section 2. This set of examples is representative of the major features and complexities of the program model that our synthesis procedure must account for, and hence make for a test bench with good coverage.

Each input specification consists of an environment process, a safety LTL specification and a liveness LTL specification. In this set of inputs, the safety and liveness specifications are fixed to False and $\Diamond\Box(\neg b)$, respectively. Each input instance is run twice, once when the synchronous synthesis tool BoSy invokes its default QBF encoding, and once when BoSy invokes our domain-specific OneHot-SAT encoding. If realizable, we output the synthesized coordinator as a CSP process. Our observations are summarized in Table 1. Our main observations are:

(1) Our implementation returned the expected outcomes for realizable specifications. The tool times out on unrealizable specifications, which is expected as the bound N (Section 3.5) for BoSy to guarantee unrealizability is very large.
(2) For all but one example (Example 4) the coordination programs obtained from both the encoding options of BoSy were identical to the ones constructed by hand in Section 2.
In Example 4, both encodings returned the same coordinator, but different from the one in Section 2. The synthesized program is *smaller* than the ones made by hand.

These observations indicate that our implementation is faithful to the theoretical development of coordination synthesis.

Table 1. Evaluation of implementation on Illustrative examples. Safety spec is False, liveness spec is $\Diamond\Box(\neg b)$.

CSP	Synthesized coordinator	Run time (in milliseconds, timeout=10s)		
		Spec. aut. construction	Sync. synthesis	
			QBF (Default)	OneHot-SAT
Example 0	$M = a_0 \rightarrow M$	138	955	86
Example 1	$M = a_0 \rightarrow M$	135	1082	89
Example 2	Unrealizable	137	Timeout	Timeout
Example 3	$M = a_0 \rightarrow M$	136	798	86
Example 4	$M = a0 \rightarrow M \mid a1 \rightarrow M$	138	773	77
Example 5	Unrealizable	138	Timeout	Timeout

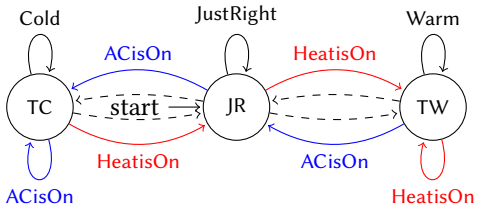

Fig. 6. Room temperature Sensor process (JR: Just right, TW: Too warm, TC: Too cold).

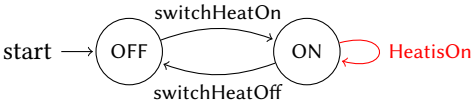

Fig. 7. Heater process.

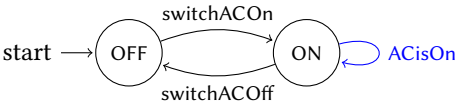

Fig. 8. AC process.

7.2.2 Thermostat: Case Study on Iterative Development. Synthesis can be thought of as a declarative paradigm for programming where a developer can focus on the *what* of the coordination problem rather than on the *how*. Through this case study we illustrate how synthesis simplifies the development of coordination programs. Our objective is to design a *smart thermostat*.

A smart thermostat interacts with a room-temperature sensor (sensor, in short), a heater, and an air-conditioner in order to maintain a comfortable room temperature. The temperature can be affected by the mode (switch-on or switch-off) of the heater and air-conditioner, and by external physical factors such as weather fluctuations, which are unpredictable and cannot be controlled. These factors prevent the smart thermostat from assessing the room temperature correctly from the modes of the devices alone. As a result, the smart thermostat must communicate with the sensor to check the room temperature, and respond accordingly to maintain ambient temperature.

CSP processes modeling the sensor, heater and air-conditioner are given in Figures 6-8. The states of the sensor denote the current temperature, while states of the heater and air-conditioner denote their mode. The dashed transitions in the sensor model fluctuations in room temperature caused by changing external physical conditions and result in private actions. The actions HeatIsOn (red transitions) and AcIsOn (blue transitions) cause private agent-to-agent interactions between the sensor and the heater or air-conditioner, respectively. Finally, the sensor communicates the current room temperature to the smart thermostat through actions Cold, JustRight, Warm, and the heater and air-conditioner interact with the smart thermostat through the Switch actions (black transitions). Therefore, the flat environment in represented by

$$\mathsf{ENV} = \mathsf{Heater}||_{\{\mathsf{HeatisOn}\}}\mathsf{Sensor}||_{\{\mathsf{ACisOn}\}}\mathsf{AC}$$

Table 2. Runtime analysis of thermostat case-study. CSP process is ENV, Safety spec is False.

Liveness spec	Synthesized coordinator	Run time (in seconds, timeout=1000s)		
		Spec. aut. construction	Sync. synthesis	
			QBF (Default)	OneHot-SAT
AmbientTemp	M = JustRight $\rightarrow M$	0.172	1.788	0.429
AmbientTemp $\wedge$ Interact	Fig 9	0.177	Timeout	40.152
AmbientTemp $\wedge$ Interact $\wedge$EnergyEfficient	Fig 10	0.188	Timeout	90.292

Its interface with the coordinator consists of public actions {JustRight, Cold, Warm, switchACOff, switchACOn, switchHeatOff, switchHeatOn}.

The safety specification is simply deadlock freedom, given by the formula False. The goal is to maintain ambient temperature. So, we begin with the following liveness specification, asserting that the temperature is infinitely often just right:

$$\text{AmbientTemp} := \Box\Diamond(\text{JustRight})$$

This coordination synthesis specification was satisfied by the trivial coordinator

$$M = \text{JustRight} \rightarrow M$$

Clearly, this coordinator relies on the private actions of the sensor (dashed transitions) to maintain ambient temperature. It never interacts with the Heater or the AC. To ensure that the synthesized coordinator interacts with the heater and AC, we append the following condition to the liveness specification:

$$\text{Interact} := \Box\Diamond\text{switchACOn} \wedge \Box\Diamond\text{switchHeatOn}$$

With the modified liveness specification, i.e., AmbientTemp $\wedge$ Interact, the coordination synthesis procedure returned a 4-processes coordinator depicted in Fig 9. It overcomes the deficiency from the previous coordinator by engaging the heater and AC infinitely often. This engagement is guaranteed due to the cycle $M_1 \xrightarrow{\text{switchACOn}} M_3 \xrightarrow{\text{switchHeatOn}} M_3 \xrightarrow{\text{switchHeatOff}} M_2 \xrightarrow{\text{JustRight}} M_1 \xrightarrow{\text{switchACOff}} M_2 \xrightarrow{\text{JustRight}} M_1$. In all executions of the coordinated system, the coordinator will be forced to visit state M_1. From here on, the coordinator is forced to take the actions as shown in the cycle. But this is still unsatisfactory, as it allows the Heater and AC to be both switched on at the same time. This can be observed along the execution $M_0 \xrightarrow{\text{JustRight}} M_1 \xrightarrow{\text{switchACOn}} M_3 \xrightarrow{\text{switchHeatOn}} M_3 \cdots$. Therefore, we add another condition to the liveness specification that enforces that the Heater and AC are not switched on at the same time:

$$\begin{aligned}\text{EnergyEfficient} :=& \neg\Diamond\Big((\text{switchACOn}) \wedge (\neg\text{switchACOff}\ \mathsf{U}\ \text{switchHeatOn})\Big)\\ &\wedge\neg\Diamond\Big((\text{switchHeatOn}) \wedge (\neg\text{switchHeatOff}\ \mathsf{U}\ \text{switchACOn})\Big)\end{aligned}$$

The coordinator obtained by modifying the liveness specification to AmbientTemp $\wedge$ Interact $\wedge$ EnergyEfficient is given in Fig 10. It is a simple 3-state coordinator. The runtime analysis has been presented in Table 2.

7.2.3 Arbiter: Case Study on Fairness. In this case study we synthesize an arbiter that allocates a shared resource to multiple concurrent processes. The arbiter must guarantee that the resource is not accessed by multiple processes at the same time, and ensure that every requesting process is eventually granted the resource. The study shows how fairness can be incorporated in specifications in order to construct appropriate solutions.

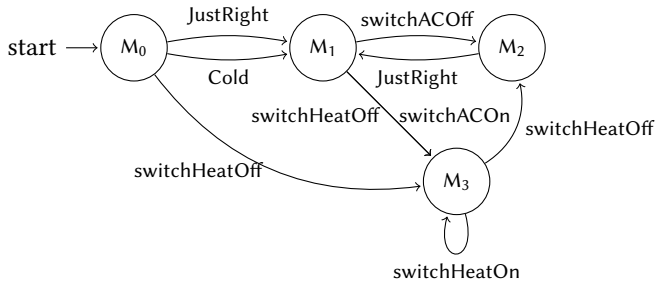

Fig. 9. Coordinator when liveness spec. is AmbientTemp ∧ Interact.

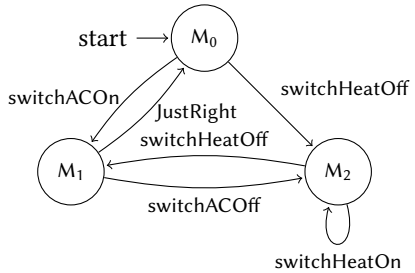

Fig. 10. Coordinator when liveness spec. is AmbientTemp ∧ Interact ∧ EnergyEfficient.

Formally, a process P(i) is defined as follows. The process cycles through *requesting* to access the common resource, waiting for the request to be *granted*, accessing the resource, and *releasing* the resource.

$$\mathsf{P(i)} = \mathsf{request.i} \rightarrow \mathsf{grant.i} \rightarrow \mathsf{release.i} \rightarrow \mathsf{P(i)}$$

For $n > 1$, let $\mathsf{ENV_n} = ||_{i \in \{0 \ldots n-1\}} \mathsf{P(i)}$ denote the environment process, which is the parallel composition of n copies of $P(i)$. Each processes synchronizes with the arbiter (coordinator) on actions request.i, grant.i, and release.i. Therefore request.i, grant.i, and release.i actions for all processes are public actions at the interface of $\mathsf{ENV_n}$ with the arbiter. As these process do not interact with each other and neither do they have private actions of their own, $\mathsf{ENV_n}$ does not have any private action and is a fully synchronous environment.

The safety specification is simply deadlock freedom, indicated by the LTL formula False. The liveness specification includes mutual exclusion and starvation freedom for every process. An arbiter can guarantee mutual exclusion if it ensures that each process must release access to the resource before another process is granted access to it:

$$\mathsf{Mutex}_n := \bigwedge_{i \in [n]} \neg\Big(\Diamond(\mathsf{grant.i} \wedge \big(\neg\mathsf{release.i}\ \mathsf{U} \bigvee_{j \neq i} \mathsf{grant.j}\big)\Big)$$

Starvation freedom is guaranteed if every requesting process is eventually granted access:

$$\mathsf{StarveFreedom}_n := \bigwedge_{i \in [n]} \Box\Big(\mathsf{request.i} \rightarrow \Diamond\mathsf{grant.i}\Big)$$

At $n = 2$, i.e., with two processes, our synthesis procedure generated the following arbiter the liveness specification given by $\mathsf{Mutex}_2 \wedge \mathsf{StarveFreedom}_2$:

$$M = \mathsf{request.0} \rightarrow \mathsf{grant.0} \rightarrow \mathsf{release.0} \rightarrow M$$

Table 3. Runtime analysis of arbiter case-study. For $n > 1$, the CSP process is $\mathsf{ENV_n}$, Safety spec is False, and Liveness spec is $\mathsf{Mutex}_n \wedge \mathsf{StarveFreedom}_n \wedge \mathsf{SimulateStrongFairness}_n$.

Number of processes (n)	Synthesized Coordinator	Run time (in seconds, timeout=1500s)		
		Spec. aut. construction	Sync. synthesis	
			QBF (Default)	OneHot-SAT
2	M_{Arbiter2}	0.166	8.255	0.6902
3	M_{Arbiter3}	0.417	1126.266	56.63

The arbiter M trivially satisfies the safety and liveness specifications on the environment process as requests from processes other than P(0) are never enabled. By construction, M is also fair w.r.t. ENV_2. Yet this is clearly not our intended coordinator. We observe that the *degree of fairness* for M may differ with respect to individual processes. Specifically, M is fair to process $P(0)$, but is only *vacuously* fair to P(1), as it never offers to synchronize with P(1).

An even stronger notion of fairness than the one assumed here would rule out such vacuous solutions, but that would require modifications to the algorithm. Instead, we modify the liveness specification by appending a condition that forces computations where the coordinator accepts a request from each process:

$$\mathsf{SimulateStrongFairness}_n := \bigwedge_{i \in [n]} \Box\Diamond \mathsf{request.i}$$

Our synthesis procedure returned the following arbiter on modifying the liveness specification to $\mathsf{Mutex}_2 \wedge \mathsf{StarveFreedom}_2 \wedge \mathsf{SimulateStrongFairness}_2$

$$M_{\mathsf{Arbiter2}} = \mathsf{request.1} \rightarrow M_1 \mid \mathsf{grant.1} \rightarrow M_{\mathsf{Arbiter2}} \mid \mathsf{release.1} \rightarrow M_{\mathsf{Arbiter2}}$$
$$M_1 = \mathsf{request.0} \rightarrow M_{\mathsf{Arbiter2}} \mid \mathsf{grant.0} \rightarrow M_1 \mid \mathsf{release.0} \rightarrow M_1$$

This solution satisfies the conditions of desired arbiter for 2-process environment and exhibits a stronger notion of fairness. In fact, it exhibits round-robin behavior despite the specification not including that as a requirement. This suggests that using higher-level models such as CSP could have advantages in generating good controllers despite under-specification.

Similar observations were made for $n = 3$. The synthesized coordination program is as follows:

$$M_{\mathsf{Arbiter3}} = \mathsf{grant.1} \rightarrow M_1 \mid \mathsf{request.1} \rightarrow M_2 \mid \mathsf{release.1} \rightarrow M_2$$
$$M_1 = \mathsf{request.2} \rightarrow M_{\mathsf{Arbiter3}} \mid \mathsf{request.1} \rightarrow M_3 \mid \mathsf{grant.1} \rightarrow M_3$$
$$M_2 = \mathsf{request.0} \rightarrow M_{\mathsf{Arbiter3}} \mid \mathsf{grant.0} \rightarrow M_2 \mid \mathsf{release.0} \rightarrow M_3$$
$$M_3 = \mathsf{grant.2} \rightarrow M_1 \mid \mathsf{release.2} \rightarrow M_2 \mid \mathsf{release.1} \rightarrow M_2 \mid \mathsf{request.2} \rightarrow M_3$$

The runtime analysis has been presented in Table 3. Unfortunately, the implementation did not scale beyond $n = 4$. But scalability is a challenge in temporal synthesis generally, e.g., recent experiments in [Faymonville et al. 2017b] on the widely-studied synchronous synthesis method show that arbiter synthesis is limited to 4-7 processes in that model.

7.2.4 Concluding Remarks. Our case studies have demonstrated the promise of synthesis in the design of coordination programs. Synthesis raises the level of abstraction and offers considerable flexibility to adjust specifications to alter or introduce requirements. Automated synthesis may even result in simpler and smaller coordination programs than those written by hand. The case studies also show, however, that the prototype implementation is limited to solving small problem

instances. We analyze why this is so, which suggests new and interesting directions for further research.

The automaton-theoretic constructions of Section 6 crucially reduce a coordination synthesis problem to a question of synchronous synthesis. Tables 1- 3 show that the time taken by this procedure is a small fraction of the total time taken by the coordination synthesis procedure. Furthermore, as the size of problem increases, the fraction of time spent inside this reduction phase decreases considerably.

The bulk of the time and space requirements are taken in the synchronous synthesis step. As described earlier in this Section, we use BoSy as the synchronous (bounded) synthesis engine. BoSy encodes a synchronous synthesis problem into a constraint-solving problem, using either SAT or QBF solvers. Although prior investigations suggest that QBF is more effective [Faymonville et al. 2017b], the domain-specific OneHot-SAT encoding handily outperforms QBF, even though the size of the encoded constraint is larger.

The scalability of the BoSy encoding is influenced by two factors: the number of states in the co-Büchi automaton given as input to BoSy, and the size of the automaton alphabet. In our experiments, the symbolic encoding of the alphabet, coupled with the OneHot-SAT encoding over the inputs keeps the second factor under control. Although the number of states of the co-Büchi automaton generated by our transformation procedure is *linear* in the size of the environment and the automaton for the LTL specification, this size appears to be the key limiting factor for the BoSy encoding (and for related tools, such as Acacia+). The limit appears to be automata with more than about 250 states.

This limit explains why the arbiter case-study did not scale well beyond 4 processes. The number of states in the flat CSP environment with 4 processes is $3^4 = 81$, and the automaton for safety are liveness specifications have 1 and 14 states, respectively. As a result, the naive construction of the universal co-Büchi automaton consists of $2 \cdot 81 \cdot 14 = 2286$ states. Even after SPOT's optimization we are left with 703 states, which is beyond the capabilities of BoSy and other synchronous synthesis solvers.

Clearly, better encodings to SAT and QBF, and improvements to the underlying synchronous synthesis tools will help for coordination synthesis. We believe, though, that a gradual improvement in back-end tools will only go so far. The most impressive impact on scalability is likely to come from entirely new methods for synthesis that utilize symmetry, modularity, and abstraction strategies to effectively handle large environment state spaces, drawing inspiration from the success of such techniques in formal verification. For instance, the arbiter environment is fully symmetric, suggesting that a combination of symmetry and modular reasoning should help reduce the complexity of synthesis. New research is needed to formulate such algorithms.

8 DISCUSSION AND RELATED WORK

The task of coordinating independent processes is one that is of importance in many domains. Coordination must work in the face of concurrency, asynchrony, partial information, noisy data, and unreliable agents. Automated synthesis methods could be of considerable help in tackling these challenges.

In this work, we have formalized the synthesis question as follows. Agents and the coordinator are modeled as processes that interact using the standard CSP handshake mechanism. The specification is given in linear temporal logic, or as a co-Büchi automaton, describing desired sequences of interactions. The main contributions are (1) in formulating an expressive model for the problem, as existing models turn out to be inadequate in various ways; (2) in the solution method, which is the first (to the best of our knowledge) to fully handle partial information, and do so for arbitrary LTL specifications; and (3) by giving a complexity-theoretic analysis of the coordination synthesis

problem. We show that there are significant practical and complexity-theoretic limitations to scalability, which suggests that research should be directed towards new methods that can handle large state spaces.

Reactive synthesis has been studied for several decades, starting with Church's formulation of this question in the 1950s (cf. [Thomas 2009]). Much of the prior work is on synthesis for the *synchronous, shared-variable* model which, as argued in the introduction, is not a good match for the coordination problems that arise in the motivating domains of multi-robot and IoT coordination. We review the relevant related work in detail below.

Reactive Synchronous Synthesis. Church's formulation of reactive synthesis was inspired by applications to hardware circuits, and thus assumes a synchronous model, where the synthesized component is a clocked Moore or Mealy machine. This question has been thoroughly studied. The tree-based view of synthesis used here originates from this line of work, in particular from the seminal results of Rabin [Rabin 1969].

The seminal work of Pnueli and Rosner [Pnueli and Rosner 1989a] on synchronous synthesis from LTL specifications was followed by the discovery of efficient solutions for the GR(1) subclass [Bloem et al. 2012; Piterman et al. 2006], "Safraless" procedures [Kupferman and Vardi 2005], and bounded synthesis methods [Filiot et al. 2010; Schewe and Finkbeiner 2007]. These methods have been implemented in a number of tools, e.g., [Bohy et al. 2012; Ehlers 2010, 2011; Faymonville et al. 2017a; Jobstmann and Roderick 2006; Pnueli et al. 2010] and applied in diverse settings (cf. [D'Ippolito et al. 2013; Kress-Gazit and Pappas 2010; Liu et al. 2013; Maoz and Sa'ar 2011, 2012]). We re-use the results of this line of research by transforming the asynchronous coordination synthesis problem to a synchronous synthesis problem, which can be solved by several of these tools.

Reactive Asynchronous Synthesis. The seminal work in asynchronous synthesis from linear temporal specifications, also for a shared-variable model, is that of Pnueli and Rosner [Pnueli and Rosner 1989b]. This model is motivated by applications to asynchronous hardware design. In the model, an adversarial scheduler chooses when the synthesized system can sample the input stream. The specification, however, has a full view of all inputs and outputs. The highly adversarial scheduler coupled with low-atomicity reads and writes makes it difficult, however, to find specifications that are realizable. Pnueli-Rosner's algorithm for asynchronous synthesis transforms the problem into a canonical synthesis problem which fits the synchronous model. That algorithm is, however, too complex for practical implementation, although work in [Klein et al. 2012] has heuristics that help solve certain cases.

In recent work [Bansal et al. 2018] present a significantly simpler and exponentially more compact construction for the asynchronous model, that has been implemented. Our method for handling hidden actions is similar to the mechanism used there to compress unobserved input values, while going beyond it to handle other forms of partial information in the richer CSP model. Indeed, one can encode the Pnueli-Rosner asynchronous problem as a coordination synthesis problem in CSP.

Schewe and Finkbeiner [Schewe and Finkbeiner 2006] model asynchronous synthesis as the design of "black-box" processes (unknowns) that interact with known "white-box" processes. There are significant differences in modeling and in the solution strategy. Their model assumes that processes are deterministic. In our model, processes may be non-deterministic, which encourages the abstraction of complex internal behavior when representing real devices or robots. A second important difference is that of the communication mechanism. In their model, communication is via individual reads and writes to shared variables. As argued in the introduction, the message-passing model of CSP permits a higher degree of atomicity and is a better fit for the motivating domains. There are other differences as well: their solution strategy is based on tree automata, as in [Pnueli and Rosner 1989a], which has proved to be difficult to implement as it requires complex

ω-automaton determinization constructions; ours is based on significantly simpler constructions that do not require determinization. d

Reactive Synthesis for CSP and similar models. Manna and Wolper were the first to consider synthesis of CSP processes from LTL specifications [Manna and Wolper 1981; Wolper 1982]; in their model, environment entities can interact only with the controller and have no hidden actions. Recent work [Ciolek et al. 2017] allows non-deterministic environments; however, all actions are visible except a distinguished τ action. The model, therefore, cannot distinguish between multiple hidden actions; neither can the specification refer to such actions. As argued in the introduction, hidden actions arise naturally from information hiding principles; our work removes both these limitations. Web services composition has been formulated in CSP-like models: in [Berardi et al. 2003], a coordinator is constructed based on a branching-time specification in Deterministic Propositional Dynamic Logic, but only in a model with full information.

The seminal work of Ramadge and Wonham [Ramadge and Wonham 1989] on discrete event control considers a control problem with restricted specifications – in particular, the only non-safety specification is that of non-blocking. The work in [Madhusudan 2001] extends the Ramadge-Wonham model with a CSP-like formulation and allows branching-time specifications, but is also based on full information about environment actions. To the best of our knowledge, this work is the first to consider partial information from hidden actions and to allow arbitrary linear-time specifications.

Process-algebraic Synthesis. In a different setting, work by Larsen, Thomsen and Liu [Larsen and Liu 1990; Larsen and Thomsen 1988] views synthesis as the solution of process algebraic inequalities of the form $E \parallel X \preceq S$ where E is the environment process, X is the unknown process, S is a process defining the specification, and $\preceq$ is a suitable process pre-order. They formulate an elegant method, using algebraic manipulations to systematically modify S based on E so that the inequality is transformed to the form $X \preceq S'$ which has a clear solution; the transformed specification S' can be viewed as a "quotient" process S/E. The construction of a quotient can incur exponential blowup, as shown in [Benes et al. 2013].

Other Synthesis Methods. Synthesis methods for shared-memory systems with branching-time specifications were developed in Emerson and Clarke's seminal work [Emerson and Clarke 1982]. Wong and Dill [Wong-Toi and Dill 1990] also consider the synthesis of a controller under synchronous and asynchronous models for shared-variable communication. In [Lustig and Vardi 2009], the authors study a different but related problem of linking together a library of finite-state machines to satisfy a temporal specification.

As discussed in Section 3, synthesis questions may also be solved by producing winning strategies for infinite games. The complexity of games with partial information is studied in [Reif 1984], while symbolic algorithms for solving such games are presented in [Raskin et al. 2007]. These game formulations, however, do not allow for asynchrony.

There is a large literature on synthesis from input-output specifications, representative results include synthesis of expressions to fill "holes" in programs [Solar-Lezama et al. 2006] and the synthesis of string transformations from examples [Harris and Gulwani 2011]. In most such instances, the synthesized program is terminating and non-reactive. One may view an LTL specification as describing an unbounded set of examples. The methods used for example-based synthesis are quite different, however; and it would be fruitful to attempt a synthesis (pun intended) of the example-driven and logic-based approaches.

ACKNOWLEDGMENTS

We would like to thank the paper and artifact reviewers for helpful suggestions. This work was supported, in part, by the National Science Foundation under Grant No. CCF-1563393. Any opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2016. Compositional synthesis of reactive controllers for multi-agent systems. In *Proc. of CAV*. Springer, 251–269.

Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. 2012. LTL to Büchi Automata Translation: Fast and More Deterministic. In *Proc. of TACAS*. 95–109.

Suguman Bansal, Kedar S. Namjoshi, and Yaniv Sa'ar. 2018. Synthesis of Asynchronous Reactive Programs from Temporal Specifications. In *Proc. of CAV*. 367–385.

Suguman Bansal, Kedar S. Namjoshi, and Yaniv Sa'ar. 2019. Synthesis of Coordination Programs from Linear Temporal Specifications. arXiv:1911.03807

Nikola Benes, Benoît Delahaye, Uli Fahrenberg, Jan Kretínský, and Axel Legay. 2013. Hennessy-Milner Logic with Greatest Fixed Points as a Complete Behavioural Specification Theory. In *Proc. of CONCUR*. 76–90.

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. 2003. Automatic Composition of E-services That Export Their Behavior. In *In proceedings of ICSOC*. 43–58.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938.

Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2012. Acacia+, a Tool for LTL Synthesis. In *Proc. of CAV*. 652–657.

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.

J. Richard Büchi and L.H. Landweber. 1969. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.* 138 (1969), 367–378.

Alonzo Church. 1957. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*. Vol. I. Cornell Univ., Ithaca, N.Y., 3–50.

Alonzo Church. 1963. Logic, arithmetic, and automata. In *Proc. Int. Congr. Math. 1962*. Inst. Mittag-Leffler, Djursholm, Sweden, 23–35.

Daniel Ciolek, Víctor A. Braberman, Nicolás D'Ippolito, Nir Piterman, and Sebastián Uchitel. 2017. Interaction Models and Automated Control under Partial Observable Environments. *IEEE Trans. Software Eng.* 43, 1 (2017), 19–33.

Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. 2013. Synthesizing nonanomalous event-based controllers for liveness goals. *Transactions on Software Engineering and Methodology* 22, 1 (2013), 9.

Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and \omega -Automata Manipulation. In *Proc. of ATVA*. 122–129.

Rüdiger Ehlers. 2010. Symbolic Bounded Synthesis. In *Proc. of CAV*. 365–379.

Rüdiger Ehlers. 2011. Unbeast: Symbolic Bounded Synthesis. In *Proc. of TACAS*. 272–275.

E Allen Emerson and Edmund M Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2, 3 (1982), 241–266.

Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. 2017b. Encodings of Bounded Synthesis. In *Proc. of TACAS*. 354–370.

Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. 2017a. BoSy: An Experimentation Framework for Bounded Synthesis. In *Proc. of CAV*. 325–332.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2009. An Antichain Algorithm for LTL Realizability. In *Proc. of CAV*. 263–277.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2010. Compositional Algorithms for LTL Synthesis. In *Proc. of ATVA*. 112–127.

Bernd Finkbeiner and Sven Schewe. 2013. Bounded synthesis. *STTT* 15, 5-6 (2013), 519–539.

Nissim Francez. 1986. *Fairness*. Springer.

William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *Symposium on Principles of Programming Languages (POPL)*, Vol. 46. 317–328.

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677.

C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

Barbara Jobstmann and Roderick. 2006. Optimizations for LTL Synthesis. In *Proc. of FMCAD*. 117–124.

Uri Klein, Nir Piterman, and Amir Pnueli. 2012. Effective Synthesis of Asynchronous Systems from GR(1) Specifications.. In *International Conference on VMCAI*. Springer, 283–298.

Hadas Kress-Gazit and George J Pappas. 2010. Automatic synthesis of robot controllers for tasks with locative prepositions. In *International Conference on Robotics and Automation (ICRA)*. IEEE, 3215–3220.

Orna Kupferman and Moshe Y Vardi. 2005. Safraless decision procedures. In *Proc. of FOCS*. IEEE, IEEE, 531–540.

Kim Guldstrand Larsen and Xinxin Liu. 1990. Equation Solving Using Modal Transition Systems. In *Proc. of LICS*.

Kim Guldstrand Larsen and Bent Thomsen. 1988. A Modal Process Logic. In *Proc. of LICS*. 203–210.

Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. 2013. Synthesis of Reactive Switching Protocols From Temporal Logic Specifications. *IEEE Trans. Automat. Contr.* 58, 7 (2013), 1771–1785.

Yoad Lustig and Moshe Y. Vardi. 2009. Synthesis from Component Libraries. In *Proc. of FOSSACS*. 395–409.

P. Madhusudan. 2001. *CONTROL AND SYNTHESIS OF OPEN REACTIVE SYSTEMS*. Ph.D. Dissertation. Institute of Mathematical Sciences, University of Madras.

Zohar Manna and Amir Pnueli. 1987. Specification and Verification of Concurrent Programs By Forall-Automata. In *Prof. of POPL*. 1–12.

Zohar Manna and Pierre Wolper. 1981. Synthesis of Communicating Processes from Temporal Logic Specifications. In *Logics of Programs, Workshop*. 253–281.

Shahar Maoz and Yaniv Sa'ar. 2011. AspectLTL: an aspect language for LTL specifications. In *In Proc. of the AOSD*. 19–30.

Shahar Maoz and Yaniv Sa'ar. 2012. Assume-Guarantee Scenarios: Semantics and Synthesis. In *Proc. of MODELS*. 335–351.

S. Moarref and H. Kress-Gazit. 2018. Reactive Synthesis for Robotic Swarms. *Formal Modeling and Analysis of Timed Systems*, 71–87.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of reactive (1) designs. In *International Conference on VMCAI*, Vol. 3855. Springer, Springer, 364–380.

Amir Pnueli. 1977. The temporal logic of programs. In *Proc. of FOCS*. IEEE, IEEE, 46–57.

Amir Pnueli and Roni Rosner. 1989a. On the Synthesis of a Reactive Module. In *Prof. of POPL*. 179–190.

Amir Pnueli and Roni Rosner. 1989b. On the synthesis of an asynchronous reactive module. *Automata, Languages and Programming* (1989), 652–671.

Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. 2010. JTLV: A Framework for Developing Verification Algorithms. In *Proc. of CAV*. 171–174.

M.O. Rabin. 1969. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.* 141 (1969), 1–35.

P.J.G. Ramadge and W.M. Wonham. 1989. The control of discrete event systems. *IEEE Transactions on Control Theory* 98 (1989).

Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. 2007. Algorithms for Omega-Regular Games with Imperfect Information. *Logical Methods in Computer Science* 3, 3 (2007).

John H. Reif. 1984. The Complexity of Two-Player Games of Incomplete Information. *J. Comput. Syst. Sci.* 29, 2 (1984), 274–301.

A. W. Roscoe. 1997. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sven Schewe and Bernd Finkbeiner. 2006. Synthesis of Asynchronous Systems. In *Proc. of LOPSTR*. 127–142.

Sven Schewe and Bernd Finkbeiner. 2007. Bounded synthesis. *Proc. of ATVA* (2007), 474–488.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *Proc. of (ASPLOS)* 34, 5 (2006), 404–415.

Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *International Conference on SAT*. 244–257.

Wolfgang Thomas. 2009. Facets of Synthesis: Revisiting Church's Problem. In *Proc. of FOSSACS*. 1–14.

Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. 2014. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Erika ÃĄbrahÃąm and Klaus Havelund (Eds.), Vol. 8413. 187–201.

Pierre Wolper. 1982. Specification and Synthesis of Communicating Processes using an Extended Temporal Logic. In *Prof. of POPL*. 20–33.

Howard Wong-Toi and David L. Dill. 1990. Synthesizing Processes and Schedulers from Temporal Specifications. In *Proc. of CAV*. 272–281.