# Distributed PROMPT-LTL Synthesis[*]

Swen Jacobs          Leander Tentrup          Martin Zimmermann

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

`lastname@react.uni-saarland.de`

We consider the synthesis of distributed implementations for specifications in Prompt Linear Temporal Logic (PROMPT–LTL), which extends LTL by temporal operators equipped with parameters that bound their scope. For single process synthesis it is well-established that such parametric extensions do not increase worst-case complexities.

For synchronous systems, we show that, despite being more powerful, the distributed realizability problem for PROMPT–LTL is not harder than its LTL counterpart. For asynchronous systems we have to consider an assume-guarantee synthesis problem, as we have to express scheduling assumptions. As asynchronous distributed synthesis is already undecidable for LTL, we give a semi-decision procedure for the PROMPT–LTL assume-guarantee synthesis problem based on bounded synthesis.

## 1   Introduction

Linear Temporal Logic [16] (LTL) is the most prominent specification language for reactive systems and the basis for industrial languages like ForSpec [2] and PSL [5]. Its advantages include a compact variable-free syntax and intuitive semantics as well as the exponential compilation property, which explains its attractive algorithmic properties: every LTL formula can be translated into an equivalent Büchi automaton of exponential size. This yields a polynomial space model checking algorithm and a doubly-exponential time algorithm for solving two-player games. Such games solve the monolithic LTL synthesis problem: given a specification, construct a correct-by-design implementation.

However, LTL lacks the ability to express timing constraints. For example, the request-response property $\mathbf{G}(req \to \mathbf{F}\,resp)$ requires that every request $req$ is eventually responded to by a $resp$. It is satisfied even if the waiting times between requests and responses diverge, i.e., it is impossible to require that requests are granted within a fixed, but arbitrary, amount of time. While it is possible to encode an a-priori fixed bound for an eventually into LTL, this requires prior knowledge of the system's granularity and incurs a blow-up when translated to automata, and is thus considered impractical.

To overcome this shortcoming of LTL, Alur et al. introduced parametric LTL (PLTL) [1], which extends LTL with parameterized operators of the form $\mathbf{F}_{\leq x}$ and $\mathbf{G}_{\leq y}$, where $x$ and $y$ are variables. The formula $\mathbf{G}(req \to \mathbf{F}_{\leq x}\,resp)$ expresses that every request is answered within an arbitrary, but fixed, number of steps $\alpha(x)$. Here, $\alpha$ is a variable valuation, a mapping of variables to natural numbers. Typically, one is interested in whether a PLTL formula is satisfied with respect to some variable valuation, e.g., model checking a transition system $\mathscr{S}$ against a PLTL specification $\varphi$ amounts to determining whether there is an $\alpha$ such that every trace of $\mathscr{S}$ satisfies $\varphi$ with respect to $\alpha$. Alur et al. showed that the PLTL model checking problem is PSPACE-complete. Due to monotonicity of the parameterized operators, one can assume that all variables $y$ in parameterized always operators $\mathbf{G}_{\leq y}$ are mapped to zero, as variable valuations are quantified existentially in the problem statements. Dually, again due to monotonicity,

one can assume that all variables $x$ in parameterized eventually operators $\mathbf{F}_{\leq x}$ are mapped to the same value, namely the maximum of the bounds. Thus, in many cases the parameterized always operators and different variables for parameterized eventually operators are not necessary.

Motivated by this, Kupferman et al. introduced PROMPT–LTL [12], which can be seen as the fragment of PLTL without the parameterized always operator and with a single bound $k$ for the parameterized eventually operators. They proved that PROMPT–LTL model checking is PSPACE-complete and solving PROMPT–LTL games is 2EXPTIME-complete, i.e., not harder than LTL games. While the results of Alur et al. rely on involved pumping arguments, the results of Kupferman et al. are all based on the so-called alternating color technique, which basically allows to reduce PROMPT–LTL to LTL. Furthermore, the result on PROMPT–LTL games was extended to PLTL games [20], again using the alternating color technique. These results show that adding parameters to LTL does not increase the asymptotic complexity of the model checking and the game-solving problem, which is still true for even more expressive logics [6, 21].

The synthesis problems mentioned above assume a setting of complete information, i.e., every part of the system has a complete view on the system as a whole. However, this setting is highly unrealistic in virtually any system. Distributed synthesis on the other hand, is the problem of synthesizing multiple components with incomplete information. Since there are specifications that are not implementable, one differentiates synthesis from the corresponding decision problem, i.e., the *realizability* problem of a formal specification. We focus on the latter, but note that from the methods presented here, implementations are efficiently extractable from a proof of realizability.

The realizability problem for distributed systems dates back to work of Pnueli and Rosner in the early nineties [17]. They showed that the realizability problem for LTL becomes undecidable already for the simple architecture of two processes with pairwise different inputs. In subsequent work, it was shown that certain classes of architectures, like pipelines and rings, can still be synthesized automatically [13, 15]. Later, a complete characterization of the architectures for which the realizability problem is decidable was given by Finkbeiner and Schewe by the *information fork* criterion [7]. Intuitively, an architecture contains an information fork, if there is an information flow from the environment to two different processes where the information to one process is hidden from the other and vice versa. The distributed realizability problem is decidable for all architectures without information fork. Beyond decidability results, semi-algorithms like bounded synthesis [8] give an architecture-independent synthesis method that is particularly well-suited for finding small-sized implementations.

**Our Contributions.** As mentioned above, one can add parameters to LTL for free: the complexity of the model checking problem and of solving infinite games does not increase. This raises the question whether this observation also holds for the distributed realizability of parametric temporal logics. For synchronous systems, we can answer this question affirmatively. For every class of architectures with decidable LTL realizability, the PROMPT–LTL realizability problem is decidable, too. To show this, we apply the alternating color technique [12] to reduce the distributed realizability problem of PROMPT–LTL to the one of LTL: one can again add parameterized operators to LTL for free.

For asynchronous systems, the environment is typically assumed to take over the responsibility for the scheduling decision [19]. Consequently, the resulting schedules may be unrealistic, e.g., one process may not be scheduled at all. While *fairness* assumptions such as "every process is scheduled infinitely often" solve this problem for LTL specifications, they are insufficient for PROMPT–LTL: a fair scheduler can still delay process activations arbitrarily long and thereby prevent the system from satisfying its PROMPT–LTL specification for any bound $k$. *Bounded fair* scheduling, where every process is guaranteed to be scheduled in bounded intervals, overcomes this problem. Since bounded fairness can be expressed in PROMPT–LTL, the realizability problem in asynchronous architectures can be formulated

more generally as an assume-guarantee realizability problem that consists of two PROMPT–LTL specifications. We give a semi-decision procedure for this problem based on a new method for checking emptiness of two-colored Büchi graphs [12] and an extension of bounded synthesis [8]. As asynchronous LTL realizability for architectures with more than one process is undecidable [19], the same result holds for PROMPT–LTL realizability. Decidability in the one process case, which holds for LTL [19], is left open.

All these results also hold for PLTL and even stronger logics [6, 21] to which the alternating color technique is still applicable.

**Related Work.** There is a rich literature regarding the synthesis of distributed systems from global $\omega$-regular specifications [4, 7, 13, 15, 17, 18]. We are not aware of work that is concerned with the realizability of parameterized logics in this setting. For local specifications, i.e., specifications that only relate the inputs and outputs of single processes, the realizability problem becomes decidable for a larger class of architectures [14]. An extension of these results to context-free languages was given by Fridman and Puchala [9]. The realizability problem for asynchronous systems and LTL specifications is undecidable for architectures with more than one process to be synthesized [19]. Later, Gastin et al. showed decidability of a restricted specification language and certain types of architectures, i.e., well-connected [11] and acyclic [10] ones. Bounded synthesis [8] provides a flexible synthesis framework that can be used for synthesizing implementations for both the asynchronous and synchronous setting.

## 2    Prompt LTL

Throughout this work, we fix a set AP of atomic propositions. The formulas of PROMPT–LTL are given by the grammar

$$\varphi ::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \varphi\,\mathbf{R}\,\varphi \mid \mathbf{F_P}\,\varphi \ ,$$

where $a \in$ AP is an atomic proposition, $\neg, \wedge, \vee$ are the usual boolean operators, and $\mathbf{X}, \mathbf{U}, \mathbf{R}$ are LTL operators next, until, and release. We use the derived operators $\mathbf{tt} := a \vee \neg a$ and $\mathbf{ff} := a \wedge \neg a$ for some fixed $a \in$ AP, and $\mathbf{F}\,\varphi := \mathbf{tt}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi := \mathbf{ff}\,\mathbf{R}\,\varphi$ as usual. Furthermore, we use $\varphi \to \psi$ as shorthand for $\neg\varphi \vee \psi$, if the antecedent $\varphi$ is a (negated) atomic proposition (where we identify $\neg\neg a$ with $a$). We define the size of $\varphi$ to be the number of subformulas of $\varphi$. The satisfaction relation for PROMPT–LTL is defined between an $\omega$-word $w = w_0 w_1 w_2 \cdots \in \left(2^{\mathrm{AP}}\right)^{\omega}$, a position $n$ of $w$, a bound $k$ for the prompt-eventually operators, and a PROMPT–LTL formula. For the LTL operators, it is defined as usual (and oblivious to $k$) and for the prompt-eventually we have

- $(w, n, k) \vDash \mathbf{F_P}\,\varphi$ if, and only if, there exists a $j$ in the range $0 \leq j \leq k$ such that $(w, n+j, k) \vDash \varphi$.

For the sake of brevity, we write $(w, k) \vDash \varphi$ instead of $(w, 0, k) \vDash \varphi$ and say that $w$ is a model of $\varphi$ with respect to $k$. Note that $(w, n, k) \vDash \varphi$ implies $(w, n, k') \vDash \varphi$ for every $k' \geq k$, i.e., satisfaction with respect to $k$ is an upwards-closed property.

**The Alternating Color Technique.**    In this subsection, we recall the alternating color technique, which Kupferman et al. introduced to solve model checking, assume-guarantee model checking, and the realizability problem for PROMPT–LTL specifications [12].

Let $r \notin$ AP be a fixed fresh proposition. An $\omega$-word $w' \in \left(2^{\mathrm{AP} \cup \{r\}}\right)^{\omega}$ is an $r$-coloring of $w \in \left(2^{\mathrm{AP}}\right)^{\omega}$ if $w'_n \cap \mathrm{AP} = w_n$, i.e., $w_n$ and $w'_n$ coincide on all propositions in AP. The additional proposition $r$ can be thought of as the color of $w'_n$: we say that the *color changes* at position $n$, if $n = 0$ or if the truth values of $r$ in $w'_{n-1}$ and in $w'_n$ are not equal. In this situation, we say that $n$ is a *change point*. An *r-block* is a maximal infix $w'_m \cdots w'_n$ of $w'$ such that the color changes at $m$ and $n+1$, but not in between.

Let $k \geq 1$: we say that $w'$ is *k-spaced* if the color changes infinitely often and each *r*-block has length at least $k$; we say that $w'$ is *k-bounded*, if each *r*-block has length at most $k$. Note that *k*-boundedness implies that the color changes infinitely often.

Given a PROMPT–LTL formula $\varphi$, let $rel_r(\varphi)$ denote the formula obtained by inductively replacing every subformula $\mathbf{F_P}\,\psi$ by

$$(r \to (r\,\mathbf{U}\,(\neg r\,\mathbf{U}\,rel_r(\psi)))) \wedge (\neg r \to (\neg r\,\mathbf{U}\,(r\,\mathbf{U}\,rel_r(\psi)))) \ ,$$

which is only linearly larger than $\varphi$ and requires every prompt eventually to be satisfied within at most one color change (not counting the position where $\psi$ holds). Furthermore, the formula $alt_r = \mathbf{GF}\,r \wedge \mathbf{GF}\,\neg r$ is satisfied if the colors change infinitely often. Finally, we define the LTL formula $c_r(\varphi) = rel_r(\varphi) \wedge alt_r$. Kupferman et al. showed that $\varphi$ and $c_r(\varphi)$ are in some sense equivalent on $\omega$-words which are bounded and spaced.

**Lemma 1** (Lemma 2.1 of [12]). *Let $\varphi$ be a* PROMPT–LTL *formula, and let* $w \in \left(2^{\mathrm{AP}}\right)^{\omega}$.

1. *If $(w, k) \vDash \varphi$, then $w' \vDash c_r(\varphi)$ for every k-spaced r-coloring $w'$ of $w$.*

2. *If $w'$ is a k-bounded r-coloring of $w$ with $w' \vDash c_r(\varphi)$, then $(w, 2k) \vDash \varphi$.*

Whenever possible, we drop the subscript $r$ for the sake of readability, if $r$ is clear from context. However, when we consider asynchronous systems in Section 4, we need to relativize two formulas with different colors, which necessitates the introduction of the subscripts.

# 3 Synchronous Distributed Synthesis

PROMPT–LTL specifications can give guarantees that LTL cannot, for example by asserting not only that requests to a system are answered *eventually*, but also that there is an *upper bound* on the reaction time. This is especially important in distributed systems, since such timing constraints become more difficult to implement because of information flows between the various parts of the system.

Consider for example a distributed computation system, where a central master gets *important* and *unimportant* tasks, and can forward tasks to a number of clients. A client can either enqueue the task, which means that it will be processed *eventually*, or clear the client-side queue and process the task immediately. The latter operation is very costly (we have to remember the open tasks as they still need to be completed), but guarantees an upper bound on the completion time. While in LTL we can only specify that all incoming tasks are processed eventually, in PROMPT–LTL we can specify that the answer time to important tasks is bounded by the formula $\mathbf{G}(important\text{-}task \to \mathbf{F_P}\,finished\text{-}task)$.[1]

We continue by formalizing the distributed realizability problem. Let $X$ and $Y$ be finite and pairwise disjoint sets of variables. A *valuation* of $X$ is a subset of $X$; thus, the set of all valuations of $X$ is $2^X$. For $w = w_0 w_1 w_2 \cdots \in (2^X)^{\omega}$ and $w' = w'_0 w'_1 w'_2 \cdots \in (2^Y)^{\omega}$, let $w \cup w' = (w_0 \cup w'_0)(w_1 \cup w'_1)(w_2 \cup w'_2) \cdots \in (2^{X \cup Y})^{\omega}$.

**Strategies.** A *strategy* $f \colon (2^X)^* \to 2^Y$ maps a history of valuations of $X$ to a valuation of $Y$. A $2^Y$-*labeled* $2^X$-*transition system* $\mathscr{S}$ is a tuple $\langle S, s_0, \Delta, l \rangle$ where $S$ is a finite set of states, $s_0 \in S$ is the designated initial state, $\Delta \colon S \times 2^X \to S$ is the transition function, and $l \colon S \to 2^Y$ is the state-labeling.

---

[1] A similar constraint could be simulated in LTL by writing that on every important incoming task, the worker queues are cleared. This, however, removes implementation freedom and requires the developer to determine how to implement the feature, instead of letting the synthesis algorithm decide.

We generalize the transition function to sequences over $2^X$ by defining $\Delta^* \colon (2^X)^* \to S$ recursively as $\Delta^*(\varepsilon) = s_0$ and $\Delta^*(w_0 \cdots w_{n-1} w_n) = \Delta(\Delta^*(w_0 \cdots w_{n-1}), w_n)$ for $w_0 \cdots w_{n-1} w_n \in (2^X)^+$. A transition system $\mathscr{S}$ *generates* the strategy $f$ if $f(w) = l(\Delta^*(w))$ for every $w \in (2^X)^*$. A strategy $f$ is called *finite-state* if there exists a transition system that generates $f$.

Let $X'$ and $Y'$ be finite and disjoint sets where $X'$ is additionally disjoint from $Y$ and $Y'$ is additionally pairwise disjoint from $X$ and $Y$. Further, let $f \colon (2^X)^* \to 2^Y$ and $f' \colon (2^X)^* \to 2^{Y'}$ be two strategies with the same domain but pairwise different co-domain $2^Y$ and $2^{Y'}$. The *product* $f \times f' \colon (2^X)^* \to 2^{Y \cup Y'}$ of $f$ and $f'$ is defined as $(f \times f')(w) = f(w) \cup f'(w)$ for every $w \in (2^X)^*$. The $2^X$-projection of a sequence $w_0 \cdots w_n \in (2^{X \cup X'})^*$ is $\mathrm{proj}_{2^X}(w_0 \cdots w_n) = (w_0 \cap X) \cdots (w_n \cap X) \in (2^X)^*$. The $2^{X'}$-widening of a strategy $f \colon (2^X)^* \to 2^Y$ is defined as $\mathrm{wide}_{2^{X'}}(f) \colon (2^{X \cup X'})^* \to 2^Y$ with $\mathrm{wide}_{2^{X'}}(f)(w) = f(\mathrm{proj}_{2^X}(w))$ for $w \in (2^{X \cup X'})^*$. For strategies $f \colon (2^X)^* \to 2^Y$ and $f' \colon (2^{X'})^* \to 2^{Y'}$, the *distributed product* $f \otimes f' \colon (2^{X \cup X'})^* \to 2^{Y \cup Y'}$ is defined as the product $\mathrm{wide}_{2^{X' \setminus X}}(f) \times \mathrm{wide}_{2^{X \setminus X'}}(f')$.

The behavior of a strategy $f \colon (2^X)^* \to 2^Y$ is characterized by an infinite tree that branches by the valuations of $X$ and whose nodes $w \in (2^X)^*$ are labeled with the strategic choice $f(w)$. For an infinite word $w = w_0 w_1 w_2 \cdots \in (2^X)^\omega$, the corresponding labeled path is defined as $(f(\varepsilon) \cup w_0)(f(w_0) \cup w_1)(f(w_0 w_1) \cup w_2) \cdots \in (2^{X \cup Y})^\omega$. We lift the set containment operator $\in$ to the containment of a labeled path $w = w_0 w_1 w_2 \cdots \in (2^{X \cup Y})^\omega$ in a strategy tree induced by $f \colon (2^X)^* \to 2^Y$, i.e., $w \in f$ if, and only if, $f(\varepsilon) = w_0 \cap Y$ and $f((w_0 \cap X) \cdots (w_i \cap X)) = w_{i+1} \cap Y$ for all $i \geq 0$. We define the satisfaction of a PROMPT–LTL formula $\varphi$ (over propositions $X \cup Y$) on strategy $f$ with respect to the bound $k$, written $(f, k) \vDash \varphi$ for short, as $(w, k) \vDash \varphi$ for all paths $w \in f$.

**Distributed Systems.** We characterize a distributed system as a set of processes with a fixed communication topology, called an *architecture* in the following. Recall that AP is the set of atomic propositions used to build formulas. An *architecture* $\mathscr{A}$ is a tuple $\langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$, where $P$ is the finite set of processes and $p_{env} \in P$ is the distinct environment process. We denote by $P^- = P \setminus \{p_{env}\}$ the set of system processes.

Given a process $p \in P$, the inputs and outputs of this process are $I_p \subseteq$ AP and $O_p \subseteq$ AP, respectively, where we assume $I_{p_{env}} = \emptyset$. We use the notation $I_{P'}$ and $O_{P'}$ for some $P' \subseteq P$ for $\bigcup_{p \in P'} I_p$ and $\bigcup_{p \in P'} O_p$, respectively. While processes may share the same inputs (in case of broadcasting), the outputs of processes must be pairwise disjoint, i.e., for all $p \neq p' \in P$ it holds that $O_p \cap O_{p'} = \emptyset$.

An *implementation* of a process $p \in P^-$ is a strategy $f_p \colon (2^{I_p})^* \to 2^{O_p}$ mapping finite input sequences to a valuation of the output variables.



(a)                                                          (b)
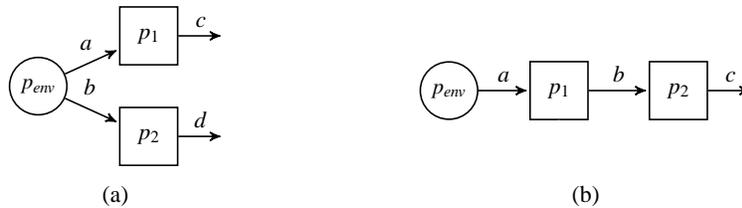
Figure 1: Example architectures

**Example 1.** Figure 1 shows example architectures $\mathscr{A}_1$ and $\mathscr{A}_2$, where

$$\mathscr{A}_1 = \langle \{p_{env}, p_1, p_2\}, p_{env}, \{p_{env} \to \emptyset, p_1 \to \{a\}, p_2 \to \{b\}\}, \{p_{env} \to \{a, b\}, p_1 \to \{c\}, p_2 \to \{d\}\} \rangle, \text{ and}$$

$$\mathscr{A}_2 = \langle \{p_{env}, p_1, p_2\}, p_{env}, \{p_{env} \to \emptyset, p_1 \to \{a\}, p_2 \to \{b\}\}, \{p_{env} \to \{a\}, p_1 \to \{b\}, p_2 \to \{c\}\} \rangle \ .$$

The architecture $\mathscr{A}_1$ in Fig. 1(a) contains two system processes, $p_1$ and $p_2$, and the environment process $p_{env}$. The processes $p_1$ and $p_2$ receive the inputs $a$, respectively $b$, from the environment and output $c$ and $d$, respectively. Hence, the environment can provide process $p_1$ with information that is hidden from $p_2$ and vice versa. In contrast, architecture $\mathscr{A}_2$, depicted in Fig. 1(b), is a pipeline architecture where information from the environment can only propagate through the pipeline processes $p_1$ and $p_2$.

**Distributed Realizability.** Let $\mathscr{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$ be an architecture. The *distributed realizability problem for $\mathscr{A}$* is to decide, given a PROMPT–LTL formula $\varphi$, whether there exist a bound $k$ and a finite-state implementation $f_p$ for every process $p \in P^-$, such that the distributed product $\bigotimes_{p \in P^-} f_p$ satisfies $\varphi$ with respect to $k$, i.e., $(\bigotimes_{p \in P^-} f_p, k) \vDash \varphi$. In this case, we say that $\varphi$ is realizable in $\mathscr{A}$. The distributed realizability problem for LTL is a special case, as LTL is a fragment of PROMPT–LTL.

Let $r \notin AP$ be the fresh proposition introduced for the alternating color technique to relativize formulas and let $\mathscr{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$ be an architecture as above. We define the architecture $\mathscr{A}^r$ as $\langle P \cup \{p_r\}, p_{env}, \{I_p\}_{p \in P} \cup \{I_r\}, \{O_p\}_{p \in P} \cup \{O_r\} \rangle$, where $I_r = \emptyset$ and $O_r = \{r\}$. Intuitively, this describes an architecture where one additional process $p_r$ is responsible for providing sequences in $(2^{\{r\}})^\omega$, i.e., a coloring by $r$. We show that $\varphi$ in $\mathscr{A}$ and $c_r(\varphi)$ in $\mathscr{A}^r$ are equi-realizable by applying the alternating color technique. As the processes are synchronized, the proof is similar to the one for the single-process case by Kupferman et al. [12].

**Theorem 1.** *A PROMPT–LTL formula $\varphi$ is realizable in $\mathscr{A}$ if, and only if, $c_r(\varphi)$ is realizable in $\mathscr{A}^r$.*

*Proof.* Let $\mathscr{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$ be an architecture and $\varphi$ be a PROMPT–LTL formula.

Assume that the PROMPT–LTL formula $\varphi$ is realizable in $\mathscr{A}$. Then, there exist finite-state strategies $f_p$ for $p \in P^-$ and a bound $k$ satisfying the PROMPT–LTL distributed realizability problem $\langle \mathscr{A}, \varphi \rangle$. For every $w \in \bigotimes_{p \in P^-} f_p$, it holds that $(w, k) \vDash \varphi$. By Lemma 1.1 it holds that every $k$-spaced $r$-coloring $w'$ of $w$ satisfies $c_r(\varphi)$. Let $f_r \colon (2^\emptyset)^* \to 2^{\{r\}}$ be a (finite-state) strategy that produces the $k$-spaced sequence $(\emptyset^k \{r\}^k)^\omega$. Then, the process implementations $\{f_p\}_{p \in P^-}$ together with $f_r$ are a solution to the LTL distributed realizability problem $\langle \mathscr{A}^r, c_r(\varphi) \rangle$.

Now, assume that the LTL formula $c_r(\varphi)$ is realizable in the architecture $\mathscr{A}^r$. Thus, there exist finite-state strategies $f_p$ for $p \in P^-$ and a finite-state strategy $f_r$ for process $p_r$. Note that the strategy $f_r \colon (2^\emptyset)^* \to 2^{\{r\}}$ has a unique output $w_r \in (2^{\{r\}})^\omega$, as it has no inputs. We claim that $w_r$ is $k$-bounded, where $k$ is the number of states of the transition system $\mathscr{S} = \langle S, s_0, \Delta, l \rangle$ generating $f_r$. To see this, note that $f_r$ has no inputs, i.e., every state of $\mathscr{S}$ has a unique successor in $\Delta$, and the unique run of $\mathscr{S}$ on $\emptyset^\omega$ ends up in a loop which is traversed ad infinitum. As the output $w_r$ has infinitely many change points, the loop contains at least one state $s$ labeled by $l(s) = \emptyset$ and at last one state $s'$ with $l(s') = \{r\}$. Thus, the maximal length of a block of $w_r$ is bounded by the length of the loop, which in turn is bounded by the size of $\mathscr{S}$.

Hence, for every $w \in \bigotimes_{p \in P^-} f_p$, the word $w_r \cup w$ is a $k$-bounded $r$-coloring of $w$ with $w_r \cup w \vDash rel_r(\varphi)$. By Lemma 1.2, for all such $w$ it holds that $(w, 2k) \vDash \varphi$. Hence, $\{f_p\}_{p \in P^-}$ together with the bound $2k$ is a solution to the PROMPT–LTL distributed realizability problem. $\square$

To conclude, we show that the newly introduced process $p_r$ preserves the *information fork* criterion [7]. Formally, consider tuples $\langle P', V', p, p' \rangle$, where $P'$ is a subset of the processes, $V'$ is a subset of the variables disjoint from $I_p \cup I_{p'}$, and $p, p' \in P^- \setminus P'$ are two different processes. Such a tuple is an information fork in $\mathscr{A}$ if $P'$ together with the edges that are labeled with at least one variable from $V'$ forms a sub-graph of $\mathscr{A}$ rooted in the environment and there exist two nodes $q, q' \in P'$ that have edges to $p, p'$, respectively, such that $O_{\{q,p\}} \not\subseteq I_{p'}$ and $O_{\{q',p'\}} \not\subseteq I_p$. For example, the architecture in Fig. 1(a)

contains the information fork $(\{p_{env}\}, \emptyset, p_1, p_2)$, while the pipeline architecture depicted in Fig. 1(b) does not contain an information fork.

**Lemma 2.** $\mathscr{A}^r$ *contains an information fork if, and only if,* $\mathscr{A}$ *contains an information fork.*

*Proof.* The *if* direction follows immediately by construction: if $\langle P', V', p, p' \rangle$ is an information fork in $\mathscr{A}$ then it is an information fork in $\mathscr{A}^r$ as well. Hence, assume $\langle P', V', p, p' \rangle$ is an information fork in $\mathscr{A}^r$. It holds that neither $p_r = p$ nor $p_r = p'$ since $p_r$ has no incoming edges. As $I_{p_r} = \emptyset$, $p_r$ cannot be in a sub-graph that is rooted in the environment, hence, $p_r \notin P'$ and $r \notin V'$. It follows that $\langle P', V', p, p' \rangle$ is an information fork in $\mathscr{A}$. □

Thus, we can use well-known results for the decidability of distributed realizability for LTL and weakly ordered architectures [7], i.e., those without an information fork.

**Corollary 1.** *Let* $\mathscr{A}$ *be an architecture. The* PROMPT–LTL *distributed realizability problem for* $\mathscr{A}$ *is decidable if, and only if,* $\mathscr{A}$ *is weakly ordered.*

Furthermore, we can directly apply semi-algorithms for the distributed realizability problem, such as bounded synthesis [8], to effectively construct small-sized solutions.

## 4  Asynchronous Distributed Synthesis

The asynchronous system model is a generalization of the synchronous model discussed in the last section. In an asynchronous system, not all processes are scheduled at the same time. We model the scheduler as part of the environment, i.e., at any given time the environment additionally signals whether a process is enabled. The resulting distributed realizability problem is already undecidable for LTL specifications and systems with more than one process [19].

We have to adapt the definition of the PROMPT–LTL realizability problem for the asynchronous setting. Using the definition from Section 3, the system can never satisfy a PROMPT–LTL formula if the scheduler is part of the environment, since it may delay scheduling indefinitely. Moreover, even if the scheduler is assumed to be fair, it can still build increasing delay blocks between process activation times, such that it is impossible for the system to guarantee any bound $k \in \mathbb{N}$. Hence, we employ the concept of *bounded fair* schedulers and allow the system valuations to depend on the scheduler bound. More generally, this is a typical instance of an assume-guarantee specification: under the assumption that the scheduler is bounded fair, the system satisfies its specification. In the following, we formally introduce the distributed realizability problem for asynchronous systems and assume-guarantee specifications.

To model scheduling, we introduce an additional set $\text{Sched} = \{sched_p \mid p \in P^-\}$ of atomic propositions. The valuation of $sched_p$ indicates whether system process $p$ is currently scheduled or not. Given a (synchronous) architecture $\mathscr{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$, we define the asynchronous architecture $\mathscr{A}^*$ as the architecture with the environment output $O^*_{p_{env}} = O_{p_{env}} \cup \text{Sched}$. Furthermore, we extend the input $I_p$ of a process by its scheduling variable $sched_p$, i.e., $I^*_p = I_p \cup \{sched_p\}$ for every $p \in P^-$. The environment can decide in every step which processes to schedule. When a process is not scheduled, its *state*—and thereby its outputs—do not change [8]. Formally, let $f_p$ for $p \in P^-$ be a finite-state implementation for a process $p$ and $\mathscr{S}_p = \langle S, s_0, \Delta, l \rangle$ a transition system that generates $f_p$. For every path $w = w_0 w_1 w_2 \cdots \in (2^{I^*_p})^\omega$ it holds that if $sched_p \notin w_i$ for some $i \in \mathbb{N}$, then $\Delta^*(w[i]) = \Delta^*(w[i+1])$, where $w[i]$ denotes the prefix $w_0 w_1 \cdots w_i$ of $w$.

A PROMPT–LTL assume-guarantee specification $\langle \varphi, \psi \rangle$ consists of a pair of PROMPT–LTL formulas. The asynchronous assume-guarantee realizability problem asks, given an asynchronous architecture $\mathscr{A}^*$ and $\langle \varphi, \psi \rangle$ as above, whether there exists a finite-state implementation $f_p$ for every process

$p \in P^-$ such that for every bound $k$ there is a bound $l$ such that for every $w \in \bigotimes_{p \in P^-} f_p$, we have that $(w, k) \vDash \varphi$ implies $(w, l) \vDash \psi$. In this case, we say that $\bigotimes_{p \in P^-} f_p$ satisfies $\langle \varphi, \psi \rangle$.

Consider the bounded fairness specification discussed above, which is expressed by the formula $\varphi = \bigwedge_{p \in P^-} \mathbf{GF_P}\, sched_p$, i.e., for every point in time, every $p$ is scheduled within a bounded number of steps. That is, we use $\varphi$ as an assumption on the environment which implies that the guarantee $\psi$ only has to be satisfied if $\varphi$ holds. Consider for example the asynchronous architecture corresponding to Fig. 1(a) and the PROMPT–LTL specification $\psi = \mathbf{G}(\mathbf{F_P}\, c \wedge \mathbf{F_P}\, \neg c \wedge \mathbf{F_P}\, d \wedge \mathbf{F_P}\, \neg d)$. Even when we assume a fair scheduler, i.e., $\varphi = \mathbf{GF}\, sched_{p_1} \wedge \mathbf{GF}\, sched_{p_2}$, the environment can prevent one process from satisfying the specification for any bound $l$. This problem is fixed by assuming the scheduler to be bounded fair, i.e., $\varphi = \mathbf{GF_P}\, sched_{p_1} \wedge \mathbf{GF_P}\, sched_{p_2}$. Then, there exist realizing implementations for processes $p_1$ and $p_2$ (that alternate between enabling and disabling the output), and the bound on the guarantee is $l = 2 \cdot k$ for every bound $k$.

Unlike LTL, where the assume-guarantee problem $\langle \varphi, \psi \rangle$ can be reduced to the LTL realizability problem for the implication $\varphi \rightarrow \psi$, this is not possible in PROMPT–LTL due to the quantifier alternation on the bounds. Indeed, it is still open whether the PROMPT–LTL assume-guarantee realizability problem in the single-process case is decidable. We show that even if the problem turns out to be decidable, an implementation that realizes the specification may need in general infinite memory.

**Lemma 3.** *There exists an assume-guarantee* PROMPT–LTL *specification that can be realized with an infinite-state strategy, but not with a finite-state strategy.*

*Proof.* Consider the assume-guarantee specification $\langle \varphi, \psi \rangle$ with $\varphi = \mathbf{GF_P}\, o \vee \mathbf{FG}\, \neg o$ and $\psi = \mathbf{ff}$ and a single process architecture with $I = \emptyset$ and $O = \{o\}$. As the guarantee $\psi$ is false, the implementation has to falsify the assumption $\varphi$ for every bound $k$ on the prompt-eventually operator to realize $\langle \varphi, \psi \rangle$. To falsify $\varphi$ with respect to $k$, the implementation has to produce a sequence $w \in (2^{\{o\}})^\omega$ where $o$ is repeatedly true and where $\emptyset^k$ is an infix of $w$. Thus, the size of the implementation depends on $k$ and an implementation that falsifies $\varphi$ for every $k$ must have infinite memory. $\square$

Since the LTL realizability problem is undecidable and implementations for PROMPT–LTL assume-guarantee specifications may need infinite memory, the PROMPT–LTL assume-guarantee realizability problem for asynchronous architectures may be at best solvable by a semi-decision procedure. We present such a semi-algorithm for the asynchronous distributed realizability problem for assume-guarantee PROMPT–LTL specifications based on bounded synthesis [8]. In bounded synthesis, a transition system of a fixed size is "guessed" and model checked by a constraint solver. Model checking for PROMPT–LTL can be solved by checking pumpable non-emptiness of colored Büchi graphs [12]. However, the pumpability condition cannot directly be expressed in the bounded synthesis constraint system. Hence, in Section 4.1, we give an alternative solution to the non-emptiness of colored Büchi graphs by a reduction to Büchi graphs that have access to the state space of the transition system. We use this result to build the semi-algorithm that is presented in Section 4.2.

## 4.1 Nonemptiness of Colored Büchi Graphs

In the case of LTL specifications, the nonemptiness problem for Büchi graphs gives a classical solution to the model checking problem for a given system $\mathscr{S}$. Let $\varphi$ be the LTL formula that $\mathscr{S}$ should satisfy. In a preprocessing step, the negation of $\varphi$ is translated to a nondeterministic Büchi word automaton $\mathscr{N}_{\neg \varphi}$ [3]. Then $\varphi$ is violated by $\mathscr{S}$ if, and only if, the Büchi graph $G$ representing the product of $\mathscr{S}$ and $\mathscr{N}_{\neg \varphi}$

is nonempty. An accepting path $\pi$ in $G$ witnesses a computation of $\mathscr{S}$ that violates $\varphi$. *Colored Büchi graphs* are an extension to those graphs in the context of model checking PROMPT–LTL [12].

A colored Büchi graph of degree two is a tuple $G = \langle \{r, r'\}, V, E, v_0, L, \mathscr{B} \rangle$ where $r$ and $r'$ are propositions, $V$ is a set of vertices, $E \subseteq V \times V$ is a set of edges, $v_0 \in V$ is the designated initial vertex, $L: V \to 2^{\{r, r'\}}$ describes the color of a vertex, and $\mathscr{B} = \{B_1, B_2\}$ is a generalized Büchi condition of index two, i.e., $B_1, B_2 \subseteq V$. A Büchi graph is a special case where we omit the labeling function and are interested in finding an accepting path. A path $\pi = v_0 v_1 v_2 \cdots \in V^\omega$ is pumpable, if we can pump all its $r'$-blocks without pumping its $r$-blocks. Formally, a path is pumpable if for all adjacent $r'$-change points $i$ and $i'$, there are positions $j$, $j'$, and $j''$ such that $i \leq j < j' < j'' < i'$, $v_j = v_{j''}$ and $r \in L(v_j)$ if, and only if, $r \notin L(v_{j'})$. A path $\pi$ is accepting, if it visits both $B_1$ and $B_2$ infinitely often. The *pumpable nonemptiness* problem for $G$ is to decide whether $G$ has a pumpable accepting path. It is NLOGSPACE-complete and solvable in linear time [12].

We give an alternative solution to this problem based on a reduction to the nonemptiness problem of Büchi graphs. To this end, we construct a non-deterministic safety automaton $\mathscr{N}_{\text{pump}}$ that characterizes the pumpability condition. Note that an infinite word is accepted by a safety automaton if, and only if, there exists an infinite run on this word.

**Lemma 4.** *Let $G = \langle \{r, r'\}, V, E, v_0, L, \mathscr{B} \rangle$ be a colored Büchi graph of degree two. There exists a Büchi graph $G'$ with $\mathscr{O}(|G'|) = \mathscr{O}(|G|^2)$ such that $G$ has a pumpable accepting path if, and only if, $G'$ has an accepting path.*

*Proof.* We define a non-deterministic safety automaton $\mathscr{N}_{\text{pump}} = \langle V \times 2^{\{r, r'\}}, S, s_0, \delta, S \rangle$ over the alphabet $V \times 2^{\{r, r'\}}$ that checks the pumpability condition. The product of $G$ and $\mathscr{N}_{\text{pump}}$ (defined later) represents the Büchi graph $G'$ where every accepting path is pumpable.

The language $\mathscr{L} \subseteq (V \times 2^{\{r, r'\}})^\omega$ of pumpable paths (with respect to a fixed set of vertices $V$) is an $\omega$-regular language that can be recognized by a small non-deterministic safety automaton. This automaton $\mathscr{N}_{\text{pump}}$ operates in 3 phases between every pair of adjacent $r'$-change points: first, it non-deterministically remembers a vertex $v$ and the corresponding truth value of $r$. Then, it checks that this value changes and thereafter it remains to show that the vertex $v$ repeats before the next $r'$-change point. Thus, the state space $S$ of $\mathscr{N}_{\text{pump}}$ is

$$\{s_0\} \cup \left\{ s_{v,x} \mid v \in V, x \in 2^{\{r, r'\}} \right\} \cup \left\{ s'_{v,y} \mid v \in V, y \in 2^{\{r, r'\}} \right\} \cup \left\{ s''_z \mid z \in 2^{\{r'\}} \right\}$$

and the initial state is $s_0$. The state space corresponds to the 3 phases: In the states $s_{v,x}$ a vertex $v$ and a truth value of $r$ are remembered, before state $s'_{v,y}$ the value of $r$ changes, and $s''_z$ is the state after the vertex repetition. The transition function $\delta: (S \times (V \times 2^{\{r, r'\}})) \to 2^S$ is defined as follows:

- $\delta(s_0, (v, x)) = \{s_{v,x}\}$

- $\delta(s_{v,x}, (v', x')) \ni \begin{cases} s_{v,x} & \text{if } x =_{\{r'\}} x' \\ s_{v',x'} & \text{if } x =_{\{r'\}} x' \\ s'_{v,x'} & \text{if } x =_{\{r'\}} x' \text{ and } x \neq_{\{r\}} x' \end{cases}$

- $\delta(s'_{v,y}, (v', x)) \ni \begin{cases} s'_{v,y} & \text{if } x =_{\{r'\}} y \text{ and } v' \neq v \\ s''_{y \cap \{r'\}} & \text{if } x =_{\{r'\}} y \text{ and } v = v \end{cases}$

- $\delta(s''_z, (v, x)) \ni \begin{cases} s''_z & \text{if } x =_{\{r'\}} z \\ s_{v,x} & \text{if } x \neq_{\{r'\}} y \end{cases}$

where $A =_C B$ is defined as $(A \cap C) = (B \cap C)$. The size of $\mathcal{N}_{\text{pump}}$ is in $O(|V|)$. Figure 2 gives a visualization of this automaton.
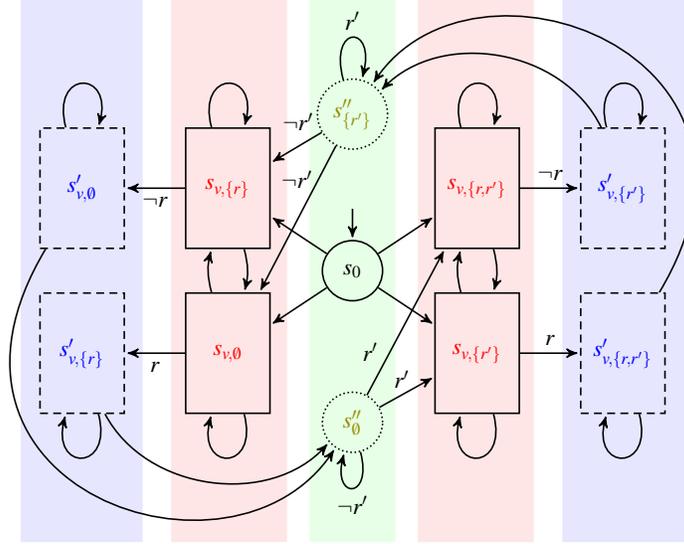


Figure 2: Schematic visualization of the automaton $\mathcal{N}_{\text{pump}}$ from the proof of Lemma 4. The 3 phases are clearly visible: In the red states $s_{v,x}$ (solid rectangles) the values $(v, x)$ are non-deterministically stored and those states can only be left if there is a change in the value of $r$. The subsequent blue states $s'_{v,y}$ (dashed rectangles) can only be left in case of a vertex repetition leading to the green state $s''_z$ (dotted circles) that waits for the next $r'$ change point.

**Remark 1.** Note that in the context of this proof, it would be enough to remember a vertex $v$ without the valuation of $\{r, r'\}$ as the vertex determines the valuation by the labeling function $L: v \to 2^{\{r,r'\}}$ of $G$. However, we will later use $\mathcal{N}_{\text{pump}}$ in a more general setting (cf. Section 4.2).

We define the product $G'$ of the colored Büchi graph $G = \langle \{r, r'\}, V, E, v_0, L, \mathscr{B} \rangle$ and the automaton $\mathcal{N}_{\text{pump}}$ as the Büchi graph $(V \times S, E', (v_0, s_0), \mathscr{B}')$, where

$$((v,s),(v',s')) \in E' \quad \Leftrightarrow \quad (v,v') \in E \wedge s' \in \delta(s, (v, L(v)))$$

and where $\mathscr{B}' = (B'_1, B'_2)$ is given by $B'_i = \{(v,s) \mid v \in B_i \text{ and } s \in S\}$ for $i \in \{1, 2\}$. The size of $G'$ is in $\mathcal{O}(|G|^2)$. It remains to show that $G$ has a pumpable accepting path if, and only if, $G'$ has an accepting path.

Consider a pumpable accepting path $\pi$ in $G$. We show that there is a corresponding accepting path $\pi'$ in $G'$. Let $i$ and $i'$ be adjacent $r'$-change points. Then there are positions $j$, $j'$, and $j''$ such that $i \leq j < j' < j'' < i'$, $v_j = v_{j''}$ and $r \in L(v_j)$ if, and only if, $r \notin L(v_{j'})$. By construction, at position $i$, automaton $\mathcal{N}_{\text{pump}}$ is some state from the set $\{s_0, s''_\emptyset, s''_{\{r'\}}\}$. We follow the automaton and remember vertex $v$ and the truth value of $r$ at position $j \geq i$ (some state $s_{v,x}$). Next, we take the transition to $s'_{v,y}$ where the truth value of $r$ changes (at position $j'$). Lastly, we check that there is a vertex repetition (at

position $j''$) and go to state $s''_z$. At the next $r'$-change point $i'$, the argument repeats. This path is accepting, as the original one is accepting.

Now, consider an accepting path $\pi$ in $G'$. We show that there is a pumpable accepting path in $G$. Let $\pi'$ be the projection of every position of $\pi$ to the first component. By construction, $\pi'$ is an accepting path in $G$. Let $\pi_i \pi_{i+1} \cdots \pi_{i'}$ be an $r'$-block of $\pi$. As $\pi$ has a run on automaton $\mathcal{N}_{\mathrm{pump}}$, we know that there exists a state repetition between $i$ and $i'$ where the truth value of $r$ changes in between. Hence, the path $\pi'$ is pumpable.                                                                                     □

## 4.2    A Semi-Algorithm for Assume-Guarantee Realizability

As the assume-guarantee realizability problem for asynchronous architectures is undecidable and infinite-state strategies are required in general, we give a semi-decision procedure for the problem, as an extension of the bounded synthesis approach [8]. Based on an LTL specification $\varphi$, an architecture $\mathscr{A}$, and a size bound $b$, bounded synthesis separately considers the problems of finding a global transition system that satisfies the given specification, and of dividing the transition system into local components according to the given architecture. To this end, two sets of constraints are generated: an encoding of the satisfaction of $\varphi$ by a global transition system $\mathscr{S}$ of size $b$, and an encoding of the architectural constraints that divides this global system into local components. If the conjunction of both sets of constraints is satisfiable, then a model of the constraints represents a distributed system that satisfies $\varphi$ in $\mathscr{A}$. Since the architectural constraints we consider are the same as in standard bounded synthesis, we only have to modify the constraints encoding the existence of a global transition system that satisfies the given specification.

In the following, we use the techniques developed in the last subsection to generalize the encoding of the specification from a single LTL formula $\varphi$ to an assume-guarantee specification $\langle \varphi, \psi \rangle$ in PROMPT–LTL. Given an assume-guarantee specification $\langle \varphi, \psi \rangle$, we first solve the problem of model-checking assume-guarantee specifications by building a universal co-Büchi tree automaton $\mathscr{U}_T$ that accepts a transition system $\mathscr{S}$ if, and only if, $\mathscr{S}$ satisfies $\langle \varphi, \psi \rangle$. From $\mathscr{U}_T$ and a given bound $b$, we then build a constraint system that is satisfiable if, and only if, an implementation $\mathscr{S}$ of $\langle \varphi, \psi \rangle$ with size $b$ exists. Finally, the encoding of architectural constraints can be adopted without changes from the original approach to obtain a conjunction of constraints that is satisfiable if, and only if, there is a system of size $b$ that satisfies $\langle \varphi, \psi \rangle$ in $\mathscr{A}$.

**Encoding $\langle \varphi, \psi \rangle$ into Büchi automata.**    Let $\mathscr{A}^* = \langle P, p_{env}, \{I_p^*\}_{p \in P}, \{O_p^*\}_{p \in P} \rangle$ be an asynchronous architecture and let $I = O_{p_{env}}^*$ and $O = \bigcup_{p \in P^-} O_p^*$ be the set of inputs, respectively outputs, of the composition of the system processes. First, we construct the non-deterministic Büchi automaton $\mathscr{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)} = \langle 2^{I \cup O \cup \{r, r'\}}, Q, q_0, \delta, B \rangle$, where $\overline{c}_{r'}(\psi) = alt_{r'} \wedge \neg rel_{r'}(\psi)$ whose language contains exactly those paths that satisfy $\overline{c}_{r'}(\psi) \wedge c_r(\varphi)$ [3].

**Lemma 5** (cf. Theorem 6.2 of [12]). *Let $\mathscr{S}$ be a $2^O$-labeled $2^I$-transition system. Then $\mathscr{S}$ does not satisfy $\langle \psi, \varphi \rangle$ if, and only if, the product of $\mathscr{S}$ and $\mathscr{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ is pumpable non-empty.*

To check the existence of pumpable error paths, we use the non-deterministic automaton $\mathscr{N}_{\mathrm{pump}} = \langle V \times 2^{\{r, r'\}}, S, s_0, \delta', S \rangle$ from the proof of Lemma 4. Here, we let $V = X \times Q$, where $X$ is a set with $b$ elements, representing the state space of the desired solution $\mathscr{S}$, and $Q$ is the state space of the automaton $\mathscr{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ defined above. That is, we use as $V$ the state space $X \times Q$ of the colored Büchi graph that is used to model check an implementation $\mathscr{S}$ against a specification $\langle \psi, \varphi \rangle$.

The product of $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ and $\mathcal{N}_{\text{pump}}$ is an automaton $\mathcal{N}$ that operates on the inputs $I$, outputs $O$, propositions $\{r, r'\}$, and the state space $X$ of the implementation, and accepts all those paths that are pumpable and violate the assume-guarantee specification (cf. Lemma 4).

$\mathcal{N}$ is defined as

$$\langle 2^{I \cup O \cup \{r, r'\}} \times X, Q \times S, (q_0, s_0), \delta^*, B^* \rangle,$$

where $\delta^* \colon Q \times S \times 2^{I \cup O \cup \{r, r'\}} \times \{x\} \to 2^{Q \times S}$ is defined as

$$\delta^*((q, s), (\sigma, x)) = \left\{ (q', s') \mid q' \in \delta(q, \sigma) \wedge s' \in \delta'(s, \{q, x\} \cup (\sigma \cap \{r, r'\})) \right\},$$

and $B^*$ is the Büchi condition $\{(q, s) \mid q \in B, s \in S\}$.

We complement $\mathcal{N}$, resulting in a universal co-Büchi automaton $\mathcal{U}$ that accepts a given sequence $w \in (2^{I \cup \{r, r'\}})^{\omega}$ of inputs and the behavior of an implementation $\mathcal{S}$ on $w$ iff the execution of $\mathcal{S}$ on $w$ satisfies $\langle \psi, \varphi \rangle$. Finally, we construct a universal co-Büchi tree automaton $\mathcal{U}_T = (2^O \times X, 2^{I \cup \{r, r'\}}, Q, q_0, \delta, \mathfrak{A})$ by spanning a copy of $\mathcal{U}$ for every direction in $2^{I \cup \{r, r'\}}$. Then, an implementation $\mathcal{S}$ is accepted by $\mathcal{U}_T$ if, and only if, $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$ (for all possible input sequences). Thus, $\mathcal{U}_T$ solves the problem of model-checking assume-guarantee specifications.

**Encoding the automaton into constraints.** Now, we use a slightly modified bounded synthesis algorithm [8] to encode $\mathcal{U}_T$ into a set of constraints in a first-order theory with uninterpreted functions and a total order, such that the constraints are satisfiable iff there exists an implementation $\mathcal{S}$ that satisfies $\langle \varphi, \psi \rangle$. The main difference to the existing approach is that the specification automaton $\mathcal{U}_T$ has access to the states of the implementation $\mathcal{S}$. This is not a problem, since the generated constraints explicitly refer to the state space of $\mathcal{S}$ anyway. The original proof of correctness can be used with minor modifications to obtain the following corollary.

**Corollary 2.** *Given an assume-guarantee specification $\langle \varphi, \psi \rangle$ and a bound b, there is a constraint system (in a decidable first-order theory) that is satisfiable if, and only if, there exist an implementation $\mathcal{S}$ of size b such that $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$.*

**Encoding of architectural constraints.** As mentioned above, the encoding of architectural constraints can be adopted without changes, and it can in particular also contain additional bounds on the state space of every single component the conjunction of both sets of constraints then asks for the existence of a distributed implementation $\mathcal{S} = \bigotimes_{p \in P^-} f_p$ of size $b$ that satisfies $\langle \varphi, \psi \rangle$, possibly with additional bounds $b_p$ for every $p \in P^-$ on the size of the components. Thus, we obtain:

**Theorem 2.** *Given an assume-guarantee specification $\langle \varphi, \psi \rangle$, an asynchronous architecture $\mathscr{A}^*$, and a family of bounds $b_p$ for every $p \in P^-$, there is a constraint system (in a decidable first-order theory) that is satisfiable if, and only if, there exist implementations $f_p$ of size $b_p$ for every $p \in P^-$ such that $\bigotimes_{p \in P^-} f_p$ satisfies $\langle \varphi, \psi \rangle$ in $\mathscr{A}^*$.*

By exhaustively traversing the space of bounds $(b_p)_{p \in P^-}$ and by solving the resulting constraint system as in the previous theorem, we obtain a semi-algorithm for the asynchronous assume-guarantee realizability problem for PROMPT–LTL. Furthermore, this also solves the synthesis problem, as implementations are efficiently obtained from a satisfying assignment of the constraint system.

**Corollary 3.** *Let $\mathscr{A}$ be an asynchronous architecture. The PROMPT–LTL distributed assume-guarantee realizability problem for $\mathscr{A}$ is semi-decidable.*

# 5   Conclusion

In this paper, we have initiated the investigation of distributed synthesis for parameterized specifications, in particular for PROMPT–LTL. This logic subsumes LTL, but additionally allows to express bounded satisfaction of system properties, instead of only eventual satisfaction. To the best of our knowledge, this is the first treatment of PROMPT–LTL specifications in distributed synthesis.

We have shown that for the case of synchronous distributed systems, we can reduce the PROMPT–LTL synthesis problem to an LTL synthesis problem. Thus, the complexity of PROMPT–LTL synthesis corresponds to the complexity of LTL synthesis, and the PROMPT–LTL realizability problem is decidable if, and only if, the LTL realizability problem is decidable. For the case of asynchronous distributed systems with multiple components, the PROMPT–LTL realizability problem is undecidable, again corresponding to the result for LTL. For this case, we give a semi-decision procedure based on a novel method for checking emptiness of two-colored Büchi graphs. All these results also hold for PLTL and the even stronger logics from [6, 21], as they have the exponential compilation property and as the alternating coloring technique is applicable to these logics as well.

Among the problems that remain open is realizability of PROMPT–LTL specifications in asynchronous distributed systems with a single component. This problem can be reduced to the (single-process) assume-guarantee realizability problem for PROMPT–LTL, which was left open in [12].

# References

[1]  Rajeev Alur, Kousha Etessami, Salvatore La Torre & Doron Peled (2001): *Parametric temporal logic for "model measuring"*. *ACM Trans. Comput. Log.* 2(3), pp. 388–407, doi:10.1145/377978.377990.

[2]  Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi & Yael Zbar (2002): *The ForSpec Temporal Logic: A New Temporal Property-Specification Language*. In Joost-Pieter Katoen & Perdita Stevens, editors: *TACAS 2002*, *LNCS* 2280, Springer, pp. 296–311, doi:10.1007/3-540-46002-0_21.

[3]  Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. The MIT Press.

[4]  Krishnendu Chatterjee, Thomas A. Henzinger, Jan Otop & Andreas Pavlogiannis (2013): *Distributed synthesis for LTL fragments*. In: *FMCAD 2013*, IEEE, pp. 18–25, doi:10.1109/FMCAD.2013.6679386.

[5]  Cindy Eisner & Dana Fisman (2006): *A Practical Introduction to PSL*. Integrated Circuits and Systems, Springer, doi:10.1007/978-0-387-36123-9.

[6]  Peter Faymonville & Martin Zimmermann (2014): *Parametric Linear Dynamic Logic*. In: *GandALF 2014*, *EPTCS* 161, pp. 60–73, doi:10.4204/EPTCS.161.8. Full version accepted for publication at *Information and Computation*.

[7]  Bernd Finkbeiner & Sven Schewe (2005): *Uniform Distributed Synthesis*. In: *LICS 2005*, IEEE Computer Society, pp. 321–330, doi:10.1109/LICS.2005.53.

[8]  Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. *STTT* 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[9]  Wladimir Fridman & Bernd Puchala (2014): *Distributed synthesis for regular and contextfree specifications*. *Acta Inf.* 51(3-4), pp. 221–260, doi:10.1007/s00236-014-0194-x.

[10]  Paul Gastin & Nathalie Sznajder (2013): *Fair Synthesis for Asynchronous Distributed Systems*. *ACM Trans. Comput. Log.* 14(2), p. 9, doi:10.1145/2480759.2480761.

[11]  Paul Gastin, Nathalie Sznajder & Marc Zeitoun (2009): *Distributed synthesis for well-connected architectures*. *Formal Methods in System Design* 34(3), pp. 215–237, doi:10.1007/s10703-008-0064-7.

[12] Orna Kupferman, Nir Piterman & Moshe Y. Vardi (2009): *From liveness to promptness*. *Formal Methods in System Design* 34(2), pp. 83–103, doi:10.1007/s10703-009-0067-z.

[13] Orna Kupferman & Moshe Y. Vardi (2001): *Synthesizing Distributed Systems*. In: *LICS 2001*, IEEE Computer Society, pp. 389–398, doi:10.1109/LICS.2001.932514.

[14] Parthasarathy Madhusudan & Pazhamaneri Subramaniam Thiagarajan (2001): *Distributed Controller Synthesis for Local Specifications*. In: *ICALP 2011*, *LNCS* 2076, Springer, pp. 396–407, doi:10.1007/3-540-48224-5_33.

[15] Swarup Mohalik & Igor Walukiewicz (2003): *Distributed Games*. In Paritosh K. Pandya & Jaikumar Radhakrishnan, editors: *FSTTCS 2003*, *LNCS* 2914, Springer, pp. 338–351, doi:10.1007/978-3-540-24597-1_29.

[16] Amir Pnueli (1977): *The temporal logic of programs*. In: *FOCS 1977*, IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[17] Amir Pnueli & Roni Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *FOCS 1990*, IEEE Computer Society, pp. 746–757, doi:10.1109/FSCS.1990.89597.

[18] Sven Schewe (2014): *Distributed synthesis is simply undecidable*. *Inf. Process. Lett.* 114(4), pp. 203–207, doi:10.1016/j.ipl.2013.11.012.

[19] Sven Schewe & Bernd Finkbeiner (2006): *Synthesis of Asynchronous Systems*. In: *LOPSTR 2006*, *LNCS* 4407, Springer, pp. 127–142, doi:10.1007/978-3-540-71410-1_10.

[20] Martin Zimmermann (2013): *Optimal bounds in parametric LTL games*. *Theor. Comput. Sci.* 493, pp. 30–45, doi:10.1016/j.tcs.2012.07.039.

[21] Martin Zimmermann (2015): *Parameterized Linear Temporal Logics Meet Costs: Still not Costlier than LTL*. In Javier Esparza & Enrico Tronci, editors: *GandALF 2015*, *EPTCS* 193, pp. 144–157, doi:10.4204/EPTCS.193.11.