Distributed Synthesis for Parameterized Temporal Logics[☆]

Swen Jacobs, Leander Tentrup, Martin Zimmermann

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

Abstract

We consider the synthesis of distributed implementations for specifications in parameterized temporal logics such as 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 distributed systems, we show that, despite being more powerful, the realizability problem for PROMPT–LTL is not harder than its LTL counterpart. For asynchronous systems, we have to express scheduling assumptions and therefore consider an assume-guarantee synthesis problem. 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. Finally, we show that our results extend to the stronger logics PLTL and PLDL.

Keywords: distributed synthesis, distributed realizability, incomplete information, parametric linear temporal logic, parametric linear dynamic logic

[☆]Supported by the projects ASDPS (JA 2357/2–1) and TriCS (ZI 1516/1–1) of the German Research Foundation (DFG) and by the grant OSARES (No. 683300) of the European Research Council (ERC).

Email addresses: `jacobs@react.uni-saarland.de` (Swen Jacobs), `tentrup@react.uni-saarland.de` (Leander Tentrup), `zimmermann@react.uni-saarland.de` (Martin Zimmermann)

1. Introduction

Linear Temporal Logic [1] (LTL) is the most prominent specification language for reactive systems and the basis for industrial languages like ForSpec [2] and PSL [3]. 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 [4]. 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 $\mathbf{G}(\mathit{req} \rightarrow \mathbf{F}\, \mathit{resp})$ 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) [5], 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 $\mathbf{G}(\mathit{req} \rightarrow \mathbf{F}_{\leq x}\, \mathit{resp})$ expresses that every request is answered within an arbitrary, but fixed, number of steps $\alpha(x)$. Here, α 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 $\mathcal{S}$ against a PLTL specification φ amounts to determining whether there is an α such that every trace of $\mathcal{S}$ satisfies φ with respect to α. Alur et al. [5] 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 [6], 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.

Intuitively, one introduces a new proposition that is thought of as coloring traces of a system. Then, one replaces each parameterized eventually operator $\mathbf{F}_{\leq x}\,\varphi$ by an LTL formula requiring φ to hold within at most one color change. If the distance between color changes is bounded from above, then satisfaction of the rewritten formula implies the existence of a bound k for the bounded eventually operators such that the original formula is satisfied with respect to k. Dually, if the distance between color changes is bounded from below, then the other implication holds: the original PROMPT–LTL formula implies the rewritten LTL formula.

When applying this equivalence, one has to specify how the truth values for the new atomic proposition coloring the traces are determined. In a game setting (in particular in synthesis), the player who aims to satisfy the PROMPT–LTL formula determines these truth values and is required to change colors infinitely often. Then, a finite-state strategy automatically ensures an upper bound on the distance between color changes.

Later, the result on PROMPT–LTL games was extended to PLTL games [7], relying on the monotonicity properties explained above and an application of 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 like Parametric Linear Dynamic Logic (PLDL) [8] and PLTL and PLDL with costs [9]. The former logic is an extension of PLTL with the full expressiveness of the ω-regular languages. The latter logics are evaluated in weighted systems and generalize PLTL and PLDL by bounding the parameterized operators in the accumulated weight instead of bounding them in time.

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 unrealistic in distributed systems. Based on this observation, *distributed synthesis* is defined as 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 typically algorithms for the realizability problem also solve the synthesis problem, as they rely on constructing implementations to prove realizability. This also holds in our work here.

The realizability problem for distributed systems dates back to work of Pnueli and Rosner in the early nineties [10]. 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 [11, 12]. 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 [13]. 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 exactly for those architectures without an information fork. Beyond decidability results, semi-decision procedures like bounded synthesis [14] give an architecture-independent synthesis method that is particularly well-suited for finding small-sized implementations. Bounded synthesis searches for finite-state implementations of a fixed size by encoding the problem as a constraint system in a decidable first-order theory. In case of a positive answer, the result is returned, otherwise the bound is increased. If there is an upper bound on the size of a finite-state implementation, then bounded synthesis is a complete decision procedure, as it can be stopped if the upper bound is reached without a positive answer. If there is no such upper bound, it is indeed a semi-decision procedure that finds an implementation if one exists, but runs forever otherwise.

1.1. 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 is also true for 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 [6] to reduce the distributed realizability problem of PROMPT–LTL to the one of LTL: one can again add parameterized operators to LTL for free. To prove this result, we add a new process whose only task it is to determine a coloring with the fresh proposition. By ensuring that the new process does not introduce an information fork we obtain decidability for the same class of architectures as for LTL.

For asynchronous systems, the environment is typically assumed to take over the responsibility for the scheduling decision [15]. 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. Hence, we have to deal with two colorings of the traces when applying the alternating color technique: One induces bounds on the parameterized eventually operators in the assumption, the other on the bounds on the parameterized eventually operators in the guarantee.

We give a semi-decision procedure for this problem based on a new method for checking emptiness of two-colored Büchi graphs [6] and an extension of bounded synthesis [14]. As asynchronous LTL realizability for architectures with more than one process is undecidable [15], the same result holds for PROMPT–LTL realizability. Thus, the semi-decision procedure is the best result one can hope for. Decidability in the one process case, which holds for LTL [15], is left open for PROMPT–LTL.

Finally, we show that all these results also hold for PLTL and PLDL, even stronger logics to which the alternating color technique and bounded synthesis are still applicable.

This is a revised and extended version of a paper that appeared at GandALF 2016 [16].

1.2. Related Work

There is a rich literature regarding the synthesis of distributed systems from global ω-regular specifications [10, 11, 12, 13, 17, 18, 19, 20]. 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 [21]. An extension of these results to context-free languages was given by Fridman and Puchala [22]. The realizability problem for asynchronous systems and LTL specifications is undecidable for architectures with more than one process to be synthesized [15]. Later, Gastin et al. showed decidability of a restricted specification language and certain types of architectures, i.e., well-connected [23] and acyclic [24] ones. Bounded synthesis [14, 25] provides a flexible synthesis framework that can be used in both the asynchronous and the synchronous setting, based on a semi-decision procedure.

1.3. Structure

In Section 2, we introduce PROMPT–LTL and the alternating color technique. In Section 3, we consider synchronous distributed synthesis for PROMPT–LTL and the asynchronous case in Section 4. Then, in Section 5, we consider both problems for the more expressive logics PLTL and PLDL. We conclude in Section 6 with a discussion of future work.

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 $a \in \mathrm{AP}$ is an atomic proposition, $\neg, \wedge, \vee$ are the usual Boolean operators, and $\mathbf{X}, \mathbf{U}, \mathbf{R}$ are the LTL operators next, until, and release. We use the derived operators $\mathbf{tt} \coloneqq a \vee \neg a$ and $\mathbf{ff} \coloneqq a \wedge \neg a$ for some fixed $a \in \mathrm{AP}$, and $\mathbf{F}\,\varphi \coloneqq \mathbf{tt}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi \coloneqq \mathbf{ff}\,\mathbf{R}\,\varphi$ as usual. Furthermore, we use $\varphi \rightarrow \psi$ as a shorthand for $\neg\varphi \vee \psi$ if the antecedent φ is an $\mathbf{F_P}$-free formula (since in that case we can transform $\neg\varphi$ into negation normal form in the fragment above). We define the size of φ to be the number of sub-fomulas of φ.

The satisfaction relation for PROMPT–LTL is defined between an ω-word $w = w_0 w_1 w_2 \cdots \in \left(2^{\mathrm{AP}}\right)^\omega$, a position $n \in \mathbb{N}$, a bound k for the prompt-eventually operators, and a PROMPT–LTL formula.

- $(w, n, k) \models a$ if, and only if, $a \in w_n$.
- $(w, n, k) \models \neg a$ if, and only if, $a \notin w_n$.
- $(w, n, k) \models \varphi_0 \wedge \varphi_1$ if, and only if, $(w, n, k) \models \varphi_0$ and $(w, n, k) \models \varphi_1$.
- $(w, n, k) \models \varphi_0 \vee \varphi_1$ if, and only if, $(w, n, k) \models \varphi_0$ or $(w, n, k) \models \varphi_1$.
- $(w, n, k) \models \mathbf{X}\, \varphi$ if, and only if, $(w, n+1, k) \models \varphi$.
- $(w, n, k) \models \varphi_0 \,\mathbf{U}\, \varphi_1$ if, and only if, there exists a $j \geq 0$ such that $(w, n+j, k) \models \varphi_1$ and $(w, n+j', k) \models \varphi_0$ for every j' in the range $0 \leq j' < j$.
- $(w, n, k) \models \varphi_0 \,\mathbf{R}\, \varphi_1$ if, and only if, for all $j \geq 0$: $(w, n+j, k) \models \varphi_1$ or $(w, n+j', k) \models \varphi_0$ for some j' in the range $0 \leq j' < j$.
- $(w, n, k) \models \mathbf{F_P}\, \varphi$ if, and only if, there exists a j in the range $0 \leq j \leq k$ such that $(w, n+j, k) \models \varphi$.

For the sake of brevity, we write $(w, k) \models \varphi$ instead of $(w, 0, k) \models \varphi$ and say that w is a model of φ with respect to k. Note that $(w, n, k) \models \varphi$ implies $(w, n, k') \models \varphi$ for every $k' \geq k$, i.e., satisfaction with respect to k is an upward-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 [6].

Let $r \notin \mathrm{AP}$ be a fixed fresh proposition. An ω-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, and 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 φ, let $rel_r(\varphi)$ denote the formula obtained by inductively replacing every sub-formula $\mathbf{F_P}\,\psi$ by

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

which is only linearly larger than φ and requires every prompt eventually to be satisfied within at most one color change (not counting the position where ψ 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 φ and $c_r(\varphi)$ are in some sense equivalent on ω-words which are bounded and spaced.

Lemma 1 (Lemma 2.1 of [6])**.** *Let φ be a* PROMPT–LTL *formula, and let $w \in \left(2^{\mathrm{AP}}\right)^{\omega}$.*

1. *If $(w, k) \models \varphi$, then $w' \models c_r(\varphi)$ for every k-spaced r-coloring w' of w.*
2. *If w' is a k-bounded r-coloring of w with $w' \models c_r(\varphi)$, then $(w, 2k) \models \varphi$.*

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 server 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 $\mathbf{G}(\textit{important-task} \rightarrow \mathbf{F_P}\,\textit{finished-task})$.[1]

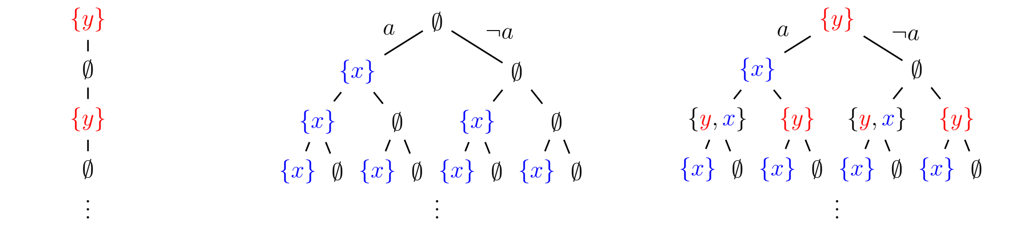
Let us now formalize the distributed realizability problem. Let X and Y be finite and pairwise disjoint sets of variables. A *valuation* of X is a subset of X; thus, the set of all valuations of X is 2^X. For $w = w_0w_1w_2\cdots \in (2^X)^\omega$ and $w' = w'_0w'_1w'_2\cdots \in (2^Y)^\omega$, let $w \cup w' = (w_0 \cup w'_0)(w_1 \cup w'_1)(w_2 \cup w'_2)\cdots \in (2^{X\cup Y})^\omega$.

Strategies. A *strategy* $f\colon (2^X)^* \rightarrow 2^Y$ maps a history of valuations of X to a valuation of Y. The behavior of a strategy $f\colon (2^X)^* \rightarrow 2^Y$ 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 $w = w_0w_1w_2\cdots \in (2^X)^\omega$, the corresponding labeled path is defined as $(f(\varepsilon)\cup w_0)(f(w_0) \cup w_1)(f(w_0w_1) \cup w_2)\cdots \in (2^{X\cup Y})^\omega$. We lift the set containment operator $\in$ to the containment of a labeled path $w = w_0w_1w_2\cdots \in (2^{X\cup Y})^\omega$ in a strategy tree induced by $f\colon (2^X)^* \rightarrow 2^Y$, 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 $i \geq 0$.

A 2^Y*-labeled* 2^X*-transition system* $\mathcal{S}$ 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, $\Delta\colon S \times 2^X \rightarrow S$ is the transition function, and $l\colon S \rightarrow 2^Y$ is the state-labeling. We generalize the transition function to sequences over 2^X by defining $\Delta^*\colon (2^X)^* \rightarrow S$ 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 $\mathcal{S}$ *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.

To reason about distributed systems, we have to combine strategies with different inputs, which we call the distributed product. To this end, we have to introduce widenings of strategies, which intuitively enlarge their domains with new atomic propositions that are ignored. Also, we need projections,

[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.



(a) Strategy f (b) Strategy g (c) Distributed product $f \otimes g$

Figure 1: Visualization of strategies $f\colon (2^{\emptyset})^* \to 2^{\{y\}}$ and $g\colon (2^{\{a\}})^* \to 2^{\{x\}}$ as infinite trees is shown in (a) and (b), respectively. The distributed product $f \otimes g$ is equal to the product of g and the $2^{\{a\}}$-widening of f and is depicted in (c).

which remove outputs from strategies.

In the following, we formally introduce these concepts. A visualization is given in Fig. 1.

Definition 1 (Distributed Product)**.** *Let X, X', Y, and Y' be finite sets such that Y and Y' are disjoint. 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 different co-domains 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 f 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 $w \in (2^{X \cup X'})^*$.*
- *Given some $g\colon (2^{X'})^* \to 2^{Y'}$, the* distributed product *$f \otimes g\colon (2^{X \cup X'})^* \to 2^{Y \cup Y'}$ is defined as the product $\mathrm{wide}_{2^{X' \setminus X}}(f) \times \mathrm{wide}_{2^{X \setminus X'}}(g)$.*

Analogously, for transition systems $\mathcal{S} = \langle S, s_0, \Delta, l\rangle$ and $\mathcal{S}' = \langle S', s_0', \Delta', l'\rangle$ the distributed product, written $\mathcal{S} \otimes \mathcal{S}'$, is defined as the transition system $\langle S \times S', (s_0, s_0'), \Delta^{\otimes}, l^{\otimes}\rangle$, where $\Delta^{\otimes}((s, s'), w) = (s'', s''')$ if, and only if, $\Delta(s, w) = s''$ and $\Delta'(s', w) = s'''$, and $l^{\otimes}(s, s') = l(s) \cup l'(s')$.

Remark 1. The strategy generated by $\mathcal{S} \otimes \mathcal{S}'$ is equal to the distributed product of the strategies generated by $\mathcal{S}$ and $\mathcal{S}'$.

We define the satisfaction of a PROMPT–LTL formula φ (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* $\mathcal{A}$ 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.

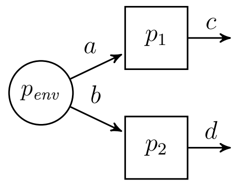
Given a process $p \in P$, the input and output signals of this process are $I_p \subseteq \text{AP}$ and $O_p \subseteq \text{AP}$, respectively, where we assume $I_{p_{env}} = \emptyset$. For $P' \subseteq P$, let $I_{P'} = \bigcup_{p \in P'} I_p$ and $O_{P'} = \bigcup_{p \in P'} O_p$. While processes 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 $O_p \cap O_{p'} = \emptyset$. Finally, we require that every input of a process originates from some other process, i.e., $I_P \subseteq O_P$.

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.

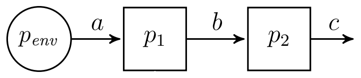
Example 1. Figure 2 shows example architectures $\mathcal{A}_1$ and $\mathcal{A}_2$:

- $\mathcal{A}_1 = \langle \{p_{env}, p_1, p_2\}, p_{env}, \{I_{p_{env}}, I_{p_1}, I_{p_2}\}, \{O_{p_{env}}, O_{p_1}, O_{p_2}\} \rangle$ with
  - $I_{p_{env}} = \emptyset, I_{p_1} = \{a\}, I_{p_2} = \{b\}$ and
  - $O_{p_{env}} = \{a, b\}, O_{p_1} = \{c\}, O_{p_2} = \{d\}$.
- $\mathcal{A}_2 = \langle \{p_{env}, p_1, p_2\}, p_{env}, \{I_{p_{env}}, I_{p_1}, I_{p_2}\}, \{O_{p_{env}}, O_{p_1}, O_{p_2}\} \rangle$ with
  - $I_{p_{env}} = \emptyset, I_{p_1} = \{a\}, I_{p_2} = \{b\}$ and
  - $O_{p_{env}} = \{a\}, O_{p_1} = \{b\}, O_{p_2} = \{c\}$.

The architecture $\mathcal{A}_1$ in Fig. 2(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 and b, respectively, 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 $\mathcal{A}_2$, depicted in Fig. 2(b), is a pipeline architecture where information from the environment can only propagate through the pipeline processes p_1 and p_2.



(a) $\mathcal{A}_1$



(b) $\mathcal{A}_2$

Figure 2: Examples for distributed architectures.

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

In the following, we show how to solve the synchronous PROMPT–LTL realizability problem. In our reduction to synchronous LTL realizability, we introduce a new process that produces a coloring sequence needed for applying the alternating color technique [6]. Let $r \notin \mathrm{AP}$ be the fresh proposition introduced for the alternating color technique to relativize formulas and let $\mathcal{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P}\rangle$ be an architecture as above. We define the architecture $\mathcal{A}^r$ 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 φ in $\mathcal{A}$ and $c_r(\varphi)$ in $\mathcal{A}^r$ 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. [6].

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

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

Assume that the PROMPT–LTL formula φ is realizable in $\mathcal{A}$. Then, there exist finite-state strategies f_p for $p \in P^-$ and a bound k satisfying the synchronous PROMPT–LTL realizability problem $\langle \mathcal{A}, \varphi \rangle$. For every $w \in \bigotimes_{p \in P^-} f_p$, it holds that $(w, k) \models \varphi$. By item 1 of Lemma 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 synchronous LTL realizability problem $\langle \mathcal{A}^r, c_r(\varphi) \rangle$.

Now, assume that the LTL formula $c_r(\varphi)$ is realizable in the architecture $\mathcal{A}^r$. 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 $\mathcal{S} = \langle S, s_0, \Delta, l \rangle$ generating f_r. To see this, note that f_r has no inputs, i.e., each state of $\mathcal{S}$ has a unique successor in Δ, and the unique run of $\mathcal{S}$ on $\emptyset^\omega$ ends up in a loop which is traversed ad infinitum. As the output w_r has infinitely many change points (since $c_r(\varphi)$ is realizable in $\mathcal{A}^r$), 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 $\mathcal{S}$.

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 $w_r \cup w \models rel_r(\varphi)$. By item 2 of Lemma 1, for all such w it holds that $(w, 2k) \models \varphi$. Hence, $\{f_p\}_{p \in P^-}$ together with the bound $2k$ is a solution to the synchronous PROMPT–LTL realizability problem. $\square$

Theorem 1 allows us to reduce the distributed realizability problem of PROMPT–LTL to the distributed realizability problem of LTL in a strategy-preserving manner as shown in the accompanying proof. In particular, we are able to reuse semi-decision procedures for the latter, such as bounded synthesis [14], to effectively construct small solutions.

To conclude, we show that the newly introduced process p_r also preserves the property whether the architecture has an *information fork* [13]. 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 $\mathcal{A}$ if P' together with the edges that are labeled with at least one variable from V' forms a subgraph of $\mathcal{A}$ 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. 2(a) contains the information fork $(\{p_{env}\}, \emptyset, p_1, p_2)$, while the pipeline architecture depicted in Fig. 2(b) has no information forks.

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

Proof. The *if* direction follows immediately by construction: if $\langle P', V', p, p' \rangle$ is an information fork in $\mathcal{A}$, then it is an information fork in $\mathcal{A}^r$ as well. Hence, assume $\langle P', V', p, p' \rangle$ is an information fork in $\mathcal{A}^r$. 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 $\mathcal{A}$. $\square$

Thus, we can use well-known results for the decidability of distributed realizability for LTL and weakly ordered architectures [13], i.e., those without an information fork.

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

4. Asynchronous Distributed Synthesis

The asynchronous system model is a generalization of the synchronous model discussed in the previous 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 [15].

We have to adapt the definition of the synchronous 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 bound 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.

Scheduling. To model scheduling, we introduce an additional set $\mathit{Sched} = \{\mathit{sched}_p \mid p \in P^-\}$ of atomic propositions. The valuation of sched_p indicates whether system process p is currently scheduled or not. Given a (synchronous) architecture $\mathcal{A} = \langle P, p_{env}, \{I_p\}_{p\in P}, \{O_p\}_{p\in P}\rangle$, we define the asynchronous architecture $\mathcal{A}^*$ as the architecture with the environment output $O^*_{p_{env}} = O_{p_{env}} \cup \mathit{Sched}$. Furthermore, we extend the input I_p of a process by its scheduling variable sched_p, i.e., $I^*_p = I_p \cup \{\mathit{sched}_p\}$ for each $p \in P^-$. The environment can decide at every step which processes to schedule. When a process is not scheduled, its *state*—and thereby its outputs—do not change [14]. Formally, for $p \in P^-$, let f_p be a finite-state strategy for a process p and $\mathcal{S}_p = \langle S, s_0, \Delta, l\rangle$ 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 $\mathit{sched}_p \notin w_i$ 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. For the remainder of this section, we will only consider such strategies.

Definition 2 (Assume-Guarantee Realizability)**.** *A* PROMPT–LTL *assume-guarantee specification* $\langle\varphi, \psi\rangle$ *consists of a pair of* PROMPT–LTL *formulas. The asynchronous* PROMPT–LTL *assume-guarantee realizability problem asks, given an asynchronous architecture* $\mathcal{A}^*$ *and* $\langle\varphi, \psi\rangle$ *as above, whether for each process* $p \in P^-$ *there exists a finite-state strategy* f_p *such that for every bound* k *on the assumption there is a bound* l *on the guarantee 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 $\varphi = \bigwedge_{p\in P^-} \mathbf{G}\mathbf{F_P}\, \mathit{sched}_p$, i.e., for every point in time, every p is scheduled within a bounded number of steps. We use φ as an assumption on the environment which implies that the guarantee ψ only has to be satisfied if φ holds. Consider, for example, the asynchronous architecture corresponding to Fig. 2(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., $\varphi = \mathbf{GF}\, \mathit{sched}_{p_1} \wedge \mathbf{GF}\, \mathit{sched}_{p_2}$, 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., $\varphi = \mathbf{G}\mathbf{F_P}\, \mathit{sched}_{p_1} \wedge \mathbf{G}\mathbf{F_P}\, \mathit{sched}_{p_2}$. 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 on the assumption.

While in the case of LTL 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. As a matter of fact, we do not know yet whether the PROMPT–LTL assume-guarantee realizability problem in the single-process case is decidable. We show that even if the problem would turn out to be decidable, an implementation that realizes the specification in general may need infinite memory.

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

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 ψ is false, the implementation has to falsify the assumption φ for every bound k on the prompt-eventually operator to realize $\langle\varphi, \psi\rangle$. To falsify φ with respect to a fixed k, the implementation has to produce a sequence $w \in (2^{\{o\}})^\omega$ where o is true infinitely often and where $\emptyset^k$ is an infix of w. Thus, the size of the implementation depends on k and an implementation that falsifies φ for every k must have infinite memory. $\square$

Moreover, already the LTL realizability problem is undecidable in the asynchronous case. Thus, 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-decision procedure for the asynchronous PROMPT–LTL assume-guarantee realizability problem based on bounded synthesis [14]. 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 [6]. 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 show how to extend bounded synthesis to such Büchi graphs in Section 4.2, and present a semi-decision procedure for PROMPT–LTL assume-guarantee synthesis based on this extension in Section 4.3.

In the following we use transition systems as representations for finite-state strategies, since the algorithm developed in this section needs access to the syntactic representation of strategies.

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 $\mathcal{S}$. Let φ be the LTL formula that $\mathcal{S}$ should satisfy. In a preprocessing step, the negation of φ is translated to a nondeterministic Büchi word automaton $\mathcal{N}_{\neg\varphi}$ [26]. Then, φ is violated by $\mathcal{S}$ if, and only if, the Büchi graph G representing the product of $\mathcal{S}$ and $\mathcal{N}_{\neg\varphi}$ is nonempty. An accepting path π in G witnesses a computation of $\mathcal{S}$ that violates φ. *Colored Büchi graphs* are an extension to such graphs in the context of model checking PROMPT–LTL [6].

A colored Büchi graph of degree two is a tuple $G = \langle \{r, r'\}, V, E, v_0, L, \mathcal{B}\rangle$, 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, $L \colon V \to 2^{\{r,r'\}}$ describes the color of a vertex, and $\mathcal{B} = \{B_1, B_2\}$ 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 $\pi = v_0v_1v_2 \cdots \in V^\omega$ 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 π 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 [6].

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 $\mathcal{N}_{\text{pump}}$ that characterizes the pumpability condition. A non-deterministic safety automaton $\mathcal{N}$ is a tuple $\langle \Sigma, S, s_0, \delta\rangle$, where Σ is a finite alphabet, S is a finite set of states, $s_0 \in S$ is the designated initial state, and $\delta \colon S \times \Sigma \to 2^S$ is the transition function. An infinite word is accepted by a safety automaton $\mathcal{N}$ if, and only if, there exists an infinite run on this word.

Lemma 4. *Let $G = \langle \{r, r'\}, V, E, v_0, L, \mathcal{B}\rangle$ be a colored Büchi graph of degree two. There exists a Büchi graph G', with $\mathcal{O}(|G'|) = \mathcal{O}(|G|^2)$, 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 $\mathcal{N}_{\text{pump}} = \langle V \times 2^{\{r,r'\}}, S, s_0, \delta\rangle$ over the alphabet $V \times 2^{\{r,r'\}}$ that checks the pumpability condition. The product of G and $\mathcal{N}_{\text{pump}}$ (defined later) represents the Büchi graph G' where every accepting path is pumpable.

The language $\mathcal{L} \subseteq (V \times 2^{\{r,r'\}})^\omega$ of pumpable paths (with respect to a fixed set of vertices V) is an ω-regular language that can be recognized by a small non-deterministic safety automaton. This automaton $\mathcal{N}_{\text{pump}}$ 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 $\mathcal{N}_{\text{pump}}$ 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 $\delta\colon (S \times (V \times 2^{\{r,r'\}})) \to 2^S$ is defined in the following. We use the notation $A =_C B$ to denote $(A \cap C) = (B \cap C)$.

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

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

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

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

The size of $\mathcal{N}_{\text{pump}}$ is in $O(|V|)$. Figure 3 gives a visualization of this automaton.

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

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

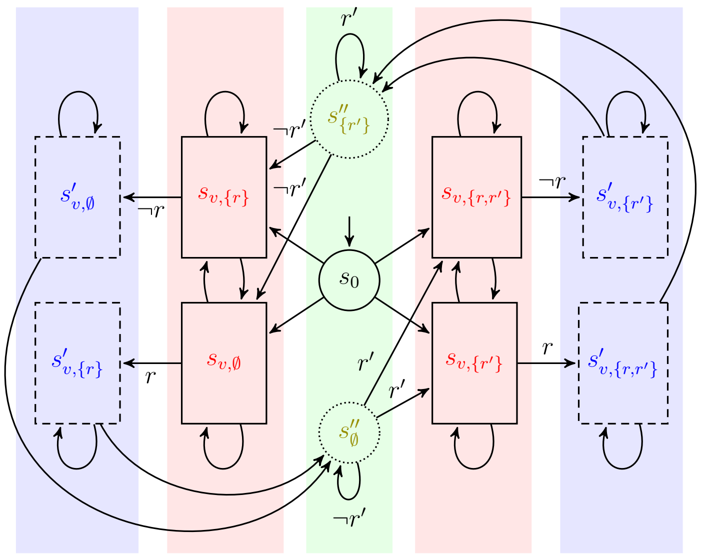


Figure 3: Schematic visualization of the automaton $\mathcal{N}_{\text{pump}}$ from the proof of Lemma 4. The 3 phases are clearly visible: In the red states $s_{v,x}$ (solid rectangles) the values (v, x) are non-deterministically stored and those states can only be left if there is a change in the value 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.

and where $\mathcal{B}' = (B'_1, B'_2)$ 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 π in G. We show that there is a corresponding accepting path π' 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 at 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 π in G'. We show that there is a pumpable accepting path in G. Let π' be the projection of every position of π to the first component. By construction, π' is an accepting path in G. Let $\pi_i \pi_{i+1} \cdots \pi_{i'}$ be an r'-block of π. As π has a run of the automaton $\mathcal{N}_{\text{pump}}$, we know that there exists a state repetition between i and i' where the truth value of r changes in between. Hence, the path π' is pumpable. □

Remark 2. Note that in the context of the previous 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 $L\colon v \to 2^{\{r,r'\}}$ of G. However, we will later use $\mathcal{N}_{\text{pump}}$ in a more general setting (cf. Section 4.3).

4.2. Bounded Synthesis

Bounded synthesis [14] is a semi-decision procedure for the distributed synthesis problem. In its original form, it takes as input a specification expressed by a universal co-Büchi automaton $\mathcal{U}$, a (possibly asynchronous) architecture $\mathcal{A}$, and a size bound b (or a family of bounds on the individual processes), and decides whether a correct implementation of the given size exists. Bounded synthesis expresses the acceptance of a transition system $\mathcal{S}$ on $\mathcal{U}$, i.e., acceptance of all traces generated by $\mathcal{S}$, as a constraint system in a decidable first-order theory. In this section, we show a modification of bounded synthesis that gives the specification automaton access to the states of the system to be synthesized. This extension is needed for automata that can express the pumpability condition, in particular the one we constructed in the proof of Lemma 4. We will show in Section 4.3 how to obtain such an automaton from a PROMPT–LTL assume-guarantee specification $\langle \varphi, \psi \rangle$, resulting in a semi-decision procedure for asynchronous distributed synthesis from this class of specifications.

For distributed architectures, bounded synthesis separately considers the problems of finding a global transition system that is accepted by $\mathcal{U}$ and of dividing the transition system into local components according to the given architecture. To this end, two sets of constraints are generated: (i) an encoding of the acceptance by $\mathcal{U}$ of a global transition system $\mathcal{S}$ of size b, and (ii) 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 satisfying assignment of the constraints represents a distributed system that satisfies φ in $\mathcal{A}$. 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.

Extended Automata. We define a *universal co-Büchi tree automaton* as a tuple $\mathcal{U} = \langle \Sigma, \Upsilon, Q, q_0, \delta, \mathfrak{A} \rangle$, where Σ is an input alphabet, Υ is a set of directions, Q is a set of states, $\delta \colon Q \times \Sigma \to 2^{Q \times \Upsilon}$ is the transition function, and $\mathfrak{A} \subseteq Q$ is the set of rejecting states.

As mentioned above, we want to check acceptance of a *global* transition system by $\mathcal{U}$. Therefore, we consider the sets of inputs $I = O^*_{p_{env}}$ and outputs $O = \bigcup_{p \in P^-} O^*_p$ of the composition of the system processes, and are interested in the acceptance of a 2^O-labeled 2^I-transition system $\mathcal{S} = \langle S, s_0, \Delta, l \rangle$. In addition, we want to recognize the pumpability condition. Therefore, we consider a *state-aware* universal co-Büchi tree automaton with $\Sigma = 2^O \times S$ and $\Upsilon = 2^I$, i.e., in addition to output valuations, the automaton has access to the current state of $\mathcal{S}$.

Acceptance of $\mathcal{S}$ by the automaton is defined in terms of run graphs: the *run graph* of an automaton $\mathcal{U}_S = \langle 2^O \times S, 2^I, Q, q_0, \delta, \mathfrak{A} \rangle$ on $\mathcal{S}$ is the minimal directed graph $\mathcal{G} = (G, E)$ that satisfies the constraints:

- $G \subseteq Q \times S$,
- $(q_0, s_0) \in G$, and
- for every $(q, s) \in G$, it holds

$$\big\{(q', \upsilon) \in Q \times 2^I \mid ((q, s), (q', \Delta(s, \upsilon))) \in E\big\} \supseteq \delta(q, (l(s), s)).$$

The *co-Büchi condition* requires that, for an infinite path $g_0 g_1 g_2 \cdots \in G^\omega$ of the run graph, $g_i \in \mathfrak{A} \times S$ holds for only finitely many $i \in \mathbb{N}$. A run graph is *accepting* if every infinite path $g_0 g_1 g_2 \cdots \in G^\omega$ of the run graph satisfies the co-Büchi condition. A transition system $\mathcal{S}$ is *accepted by* $\mathcal{U}_S$ if the (unique) run graph of $\mathcal{U}_S$ on $\mathcal{S}$ is accepting.

Annotated transition systems. We introduce an annotation function for transition systems that witnesses acceptance by a (possibly state-aware) universal co-Büchi tree automaton. The annotation assigns to each pair $(q, s) \in Q \times S$ a natural number or a special symbol $\bot$. Natural numbers indicate the maximal number of occurrences of rejecting states on any path to (q, s) in the run graph; $\bot$ indicates that the pair (q, s) is not reachable. Thus, if for a given transition system there exists an annotation that assigns natural numbers to all vertices of the run graph, then the number of visits to rejecting states must be bounded in any run. Such annotations are called *valid*, and transition systems with valid annotations are exactly those that are accepted by the automaton.

An *annotation* of a 2^O-labeled 2^I-transition system $\mathcal{S} = \langle S, s_0, \Delta, l\rangle$ on a state-aware universal co-Büchi tree automaton $\mathcal{U}_S = \langle 2^O \times S, 2^I, Q, q_0, \delta, \mathfrak{A}\rangle$ is a function $\lambda\colon Q \times S \to \{\bot\} \cup \mathbb{N}$. An annotation is *valid* if it satisfies the following conditions:

- $\lambda(q_0, s_0) \neq \bot$
- for any $(q, s) \in Q \times S$:

  if $\lambda(q, s) = n \neq \bot$ and $(q', \upsilon) \in \delta(q, (l(s), s))$
  then $\lambda(q', \Delta(s, \upsilon)) \rhd \lambda(q, s)$,
  where $\rhd$ is interpreted as $>$ if $q' \in \mathfrak{A}$, and $\geq$ otherwise.

An annotation is *c-bounded* if its codomain is contained in $\{\bot, 0, \ldots, c\}$.

Theorem 2 (see [14])**.** *A finite-state O-labeled I-transition system $\mathcal{S} = \langle S, s_0, \Delta, l\rangle$ is accepted by a state-aware universal co-Büchi tree automaton $\mathcal{U}_S = \langle 2^O \times S, 2^I, Q, q_0, \delta, \mathfrak{A}\rangle$ if, and only if, it has a valid $(|S| \cdot |\mathfrak{A}|)$-bounded annotation.*

Proof. The original proof by Finkbeiner and Schewe [14] works without modifications for our extension to state-aware universal co-Büchi tree automata. $\square$

For a given state-aware universal co-Büchi tree automaton $\mathcal{U}_S = \langle 2^O \times S, 2^I, Q, q_0, \delta, \mathfrak{A}\rangle$, Theorem 2 allows us to decide the existence of an O-labeled I-transition system with state space S that is accepted by $\mathcal{U}_S$.

Encoding of global acceptance. The existence of a (global) transition system with a valid annotation can be encoded into a set of decidable constraints in first-order logic modulo a theory with uninterpreted functions and a partial order. Essentially, we can directly encode the conditions for a valid annotation into constraints, with uninterpreted transition function and labeling for the desired transition system. Such constraints can then be solved by off-the-shelf satisfiability modulo theories (SMT) tools.

Like the proof of Theorem 2, the original encoding can easily be extended to support our notion of state-aware universal Büchi tree automata. It is constructed in the following way:

1. Assume that $\mathcal{U}_S$ is defined in a suitable way, i.e., the sets Q and $\mathfrak{A}$, state q_0, and transition relation $\delta\colon Q \times (2^O \times S) \to 2^{(Q \times 2^I)}$ are defined.
2. Declare uninterpreted sets and functions for the transition system $\mathcal{S}$ and the annotation:
   - Define the set of states S as $\{1, \dots, b\}$ for the given bound $b \in \mathbb{N}$.
   - Declare the transition function of $\mathcal{S}$ as $\Delta : S \times 2^I \to S$ and the labeling function as $l : S \to 2^O$.
   - Declare two functions that are used to model the annotation function: $\lambda^{\mathbb{B}} : Q \times S \to \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$ and $\lambda^{\#} : Q \times S \to \mathbb{N}$.
3. Assert the following constraints:

$$\begin{aligned}
& s_0 \in S \\
& \lambda^{\mathbb{B}}(q_0, s_0) \\
\forall q, q' \in Q, s \in S, \upsilon \in 2^I : \quad & \lambda^{\mathbb{B}}(q, s) \land (q', \upsilon) \in \delta(q, (l(s), s)) \\
& \to \lambda^{\mathbb{B}}(q', \Delta(s, \upsilon)) \land \lambda^{\#}(q', \Delta(s, \upsilon)) \geq \lambda^{\#}(q, s) \\
\forall q, q' \in Q, s \in S, \upsilon \in 2^I : \quad & \lambda^{\mathbb{B}}(q, s) \land (q', \upsilon) \in \delta(q, (l(s), s)) \land q' \in \mathfrak{A} \\
& \to \lambda^{\#}(q', \Delta(s, \upsilon)) > \lambda^{\#}(q, s)
\end{aligned}$$

The encoding ensures that $\lambda^{\mathbb{B}}(q, s)$ is true whenever $(q, s) \in Q \times S$ is reachable in the run graph of $\mathcal{U}_S$ on $\mathcal{S}$, and that $\lambda^{\#}$ respects the conditions for a valid annotation for all reachable vertices (q, s). Since there are no conditions for the annotation on vertices that are not reachable, a solution for $\lambda^{\#}$ will represent a valid annotation of $\mathcal{S}$ on $\mathcal{U}_S$.

Note that our encoding is a strict generalization of the encoding of Finkbeiner and Schewe [14]. In particular, our encoding can also be used for specifications in LTL that are translated into a universal co-Büchi tree automaton (see Kupferman and Vardi [27]), which can be seen as a state-aware automaton that ignores the state of the transition system.

Encoding of architectural constraints. As mentioned above, the encoding of architectural constraints can be adopted from the original approach without changes. For a given asynchronous architecture $\mathcal{A}^* = \langle P, p_{env}, \{I_p^*\}_{p \in P}, \{O_p^*\}_{p \in P}\rangle$, the additional constraints (1) assert that the state of a process $p \in P^-$ does not change if it is not scheduled and (2) that the transitions of a process only depend on its current state and the visible inputs. In addition, it can 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 $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$ of size b that is accepted by $\mathcal{U}$, possibly with additional bounds b_p for every $p \in P^-$ on the size of the components.

Theorem 3 (see [14])**.** *Given a state-aware universal co-Büchi tree automaton $\mathcal{U}_S$* [2]*, an asynchronous architecture $\mathcal{A}^*$, 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 $\mathcal{S}_p$ of size b_p for every $p \in P^-$ such that $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$ is accepted by $\mathcal{U}_S$ and satisfies the architectural constraints of $\mathcal{A}^*$.*

Proof. Follows immediately from Theorem 2 and the correctness of the architectural constraints from Finkbeiner and Schewe [14]. □

4.3. A Semi-Decision Procedure for Assume-Guarantee Realizability

Since 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. Our solution is based on the techniques developed in the last subsections.

As the bounded synthesis approach described in the last subsection already accounts for "guessing" transition systems $\mathcal{S}_p$ for each system process p according to the architectural constraints given by $\mathcal{A}^*$, we reduce the problem of model checking individual implementations $\mathcal{S}_p$ to model checking the product system $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$. A transition system $\mathcal{S}$ satisfies an assume-guarantee specification $\langle \varphi, \psi \rangle$ if the strategy f generated by $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$, i.e., if for every bound k there is a bound l such that for every $w \in f$, we have that $(w, k) \vDash \varphi$ implies $(w, l) \vDash \psi$.

[2]As the symbol S in $\mathcal{U}_S$ refers to the state-space of the distributed product, $|S|$ has to be equal to the product of bounds b_p for $p \in P^-$.

Given an assume-guarantee specification $\langle \varphi, \psi \rangle$, we first solve the problem of model checking assume-guarantee specifications by building a state-aware universal co-Büchi tree automaton $\mathcal{U}_S$ that accepts a transition system $\mathcal{S}$ if, and only if, $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$. Given $\mathcal{U}_S$ and a bound b on the size of the implementation, we can then use the encoding from Section 4.2 to decide realizability modulo this bound, and obtain a semi-decision procedure by solving the problem for increasing bounds.

Encoding $\langle \varphi, \psi \rangle$ into Büchi automata. Let $\mathcal{A}^* = \langle P, p_{env}, \{I^*_p\}_{p \in P}, \{O^*_p\}_{p \in P} \rangle$ be an asynchronous architecture and let $I = O^*_{p_{env}}$ and $O = \bigcup_{p \in P^-} O^*_p$ be the set of inputs and outputs, respectively, of the composition of the system processes. First, we construct the non-deterministic Büchi automaton $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)} = \langle 2^{I \cup O \cup \{r,r'\}}, Q, q_0, \delta, B \rangle$, where $\overline{c}_{r'}(\psi) = alt_{r'} \wedge \neg rel_{r'}(\psi)$, whose language contains exactly those paths that satisfy $\overline{c}_{r'}(\psi) \wedge c_r(\varphi)$ [26]. Then, we use the following lemma to characterize whether a transition system $\mathcal{S}$ satisfies an assume-guarantee specification $\langle \varphi, \psi \rangle$ by reducing it to finding pumpable error paths in the two-color Büchi graph $G = \langle \{r, r'\}, V, E, v_0, L, \mathcal{B} \rangle$, as introduced in Section 4.1, which is the product of $\mathcal{S} = \langle S, s_0, \Delta, l \rangle$ and $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$. Formally, the elements of G are defined as $V = S \times 2^{\{r,r'\}} \times Q$, E as $((s, R, q), (s', R', q')) \in E$ if, and only if, there is an input valuation $\vec{i} \in 2^I$ such that $s' = \Delta(s, \vec{i})$ and $(q', \vec{i}) \in \delta(q, l(s))$, $v_0 = (s_0, \emptyset, q_0)$, L as $L((s, R, q, q^*)) = R$, and $\mathcal{B} = \{B\}$.

Lemma 5. *Let $\langle \varphi, \psi \rangle$ be a* PROMPT–LTL *assume-guarantee specification, $\mathcal{A}^*$ an asynchronous architecture and $\mathcal{S}_p$ a finite-state implementation for every system process $p \in P^-$. The distributed product $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$ does not satisfy $\langle \varphi, \psi \rangle$ if, and only if, the product of $\mathcal{S}$ and $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ is pumpable non-empty.*

Proof. Similar to the proof of Lemma 6.1 and Theorem 6.2 in [6]. The missing proof of Lemma 6.1 is given in [8] (Lemma 8). See also the discussion below its proof. □

To check the existence of pumpable error paths, we use the non-deterministic automaton $\mathcal{N}_{\text{pump}} = \langle V \times 2^{\{r,r'\}}, S, s_0, \delta', S \rangle$ 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 $\mathcal{S}$, and Q is the state space of the automaton $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ 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 $\mathcal{S}$ 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).

Formally, $\mathcal{N}$ 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

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

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 if, and only if, the execution of $\mathcal{S}$ on w satisfies $\langle\psi,\varphi\rangle$. Finally, we construct a (state-aware) universal co-Büchi tree automaton $\mathcal{U}_S = (2^O \times X, 2^{I\cup\{r,r'\}}, Q, q_0, \delta, \mathfrak{B})$ by spanning a copy of $\mathcal{U}$ for every direction in $2^{I\cup\{r,r'\}}$. Then, an implementation $\mathcal{S}$ with set S of states is accepted by $\mathcal{U}_S$ if, and only if, $\mathcal{S}$ satisfies $\langle\varphi,\psi\rangle$ (for all possible input sequences). Thus, $\mathcal{U}_S$ solves the problem of model checking assume-guarantee specifications.

Encoding the automaton into constraints. Now, we can use the modified bounded synthesis algorithm from Section 4.2 to encode $\mathcal{U}_S$ into a set of constraints that is satisfiable if, and only if, there exists an implementation $\mathcal{S}$ that satisfies $\langle\varphi,\psi\rangle$. We obtain the following corollaries stating the correctness of the constraint system for single-process implementations (Corollary 2) and distributed implementations (Corollary 3), respectively.

Corollary 2. *Given a* PROMPT–LTL *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 exists an implementation* $\mathcal{S}$ *of size* b *such that* $\mathcal{S}$ *satisfies* $\langle\varphi,\psi\rangle$*.*

Corollary 3. *Given a* PROMPT–LTL *assume-guarantee specification* $\langle\varphi,\psi\rangle$*, an asynchronous architecture* $\mathcal{A}^*$*, and a family of bounds* b_p *for each* $p \in P^-$*, there is a constraint system (in a decidable first-order theory) that is satisfiable if, and only if, there exist implementations* $\mathcal{S}_p$ *of size* b_p *for each* $p \in P^-$ *such that* $\bigotimes_{\mathcal{S}\in P^-} \mathcal{S}_p$ *satisfies* $\langle\varphi,\psi\rangle$ *in* $\mathcal{A}^*$*.*

By exhaustively traversing the space of bounds $(b_p)_{p \in P^-}$ and by solving the resulting constraint system, we obtain a semi-decision procedure for the asynchronous PROMPT–LTL assume-guarantee realizability problem. Furthermore, this also solves the synthesis problem, as a satisfying assignment of the constraint system directly represents a valid implementation, where the transition relation is given by (the assignment of) the function Δ, and the state labeling by (the assignment of) the function l.

Corollary 4. *Let $\mathcal{A}^*$ be an asynchronous architecture. The asynchronous* PROMPT–LTL *assume-guarantee realizability problem for $\mathcal{A}^*$ is semi-decidable.*

5. Beyond PROMPT–LTL

In this section, we consider distributed synthesis for logics stronger than PROMPT–LTL. As already pointed out in the introduction, PROMPT–LTL is predated by parametric linear temporal logic (PLTL), which was introduced by Alur et al. [5]. This logic is obtained by adding parameterized eventually operators of the form $\mathbf{F}_{\leq x}\,\varphi$ and parameterized always operators of the form $\mathbf{G}_{\leq y}$ to LTL. Here, x and y are variables which are instantiated by a variable valuation α mapping variables to natural numbers that serve as bounds: $\mathbf{F}_{\leq x}\,\varphi$ holds with respect to α if φ holds within the next $\alpha(x)$ steps, while $\mathbf{G}_{\leq y}\,\varphi$ holds with respect to α, if φ holds at least for the next $\alpha(y)$ steps. Thus, intuitively, the variables bound the scope of the operators. In particular, PROMPT–LTL can be seen as the fragment of PLTL without parameterized always operators and where all parameterized eventually operators are parameterized by the same variable.

Alur et al. showed that the model checking problem for PLTL, where the variable valuation α is existentially quantified, is PSpace-complete, and therefore not harder than LTL model checking. Later, a similar result was shown for solving infinite games with PLTL winning conditions, which is still complete for doubly-exponential time [7]. As for PROMPT–LTL, distributed synthesis for PLTL specifications has never been considered before.

The second logic we consider in this section is parametric linear dynamic logic (PLDL) [8], which has its roots in another shortcoming of LTL: it lacks the full expressive power of the ω-regular languages. There is a long line of extensions of LTL addressing this issue [28, 4, 29]. Most recently, Vardi introduced linear dynamic logic (LDL), which adds regular expressions as *guards* to the temporal operators of LTL: the formula $\langle g \rangle\,\varphi$ holds if there is a position such that the prefix up to it matches the guard g and φ holds at this position. Similarly, $[g]\,\varphi$ holds, if φ holds at all positions where the prefix up to it matches the guard. Thus, the diamond operator is a guarded eventually operator and the box operator is a guarded always operator. Vardi showed that LDL has the exponential compilation property [30], i.e., formulas can be translated into equivalent Büchi automata of exponential size. Thus, LDL model checking is still PSPACE-complete while solving LDL games is still 2EXPTIME-complete.

Now, PLDL is obtained by allowing parameterized diamond and box operators, with the expected semantics. For the first time, this logic addresses both shortcomings of LTL, lack of timing constraints and limited expressiveness, simultaneously. Even in this setting, model checking is just PSPACE-complete and solving games is 2EXPTIME-complete [8]. Distributed synthesis for PLDL specifications has never been considered before.

In this section, we address the distributed synthesis problem for both logics, starting with the synchronous variant. For PLTL, we rely on a reduction to the PROMPT–LTL synthesis problem. The variable valuation α will be existentially quantified in the problem statement, just as the bound k in the case of PROMPT–LTL synthesis is existentially quantified. Now, consider a parameterized always operator $\mathbf{G}_{\leq y}\,\varphi$: if φ is satisfied for at last $\alpha(y)$ steps, then also for at least zero steps, i.e., at the current position. Thus, when the value for y is existentially quantified, $\mathbf{G}_{\leq y}\,\varphi$ degenerates to the formula φ, as y can always be instantiated with 0.

Dually, consider a parameterized eventually operator $\mathbf{F}_{\leq x}\,\varphi$: if φ holds at least once within the next $\alpha(x)$ steps, then also at least once within the next k steps, for every $k \geq \alpha(x)$. Thus, if α is existentially quantified, then one can replace all variables parameterizing parameterized eventually operators by a unique one. By applying these two replacements, one obtains an equivalent PROMPT–LTL formula, provided α is existentially quantified. In fact, these observations were the impetus to introduce PROMPT–LTL. However, the situation is different when one is interested in a fixed variable valuation or for optimization problems. In this case, the replacements are no longer valid.

Then, we consider the synchronous synthesis problem for PLDL, which we solve along the same lines as for its special case PROMPT–LTL: the alternating color technique has been reformulated for PLDL and the exponential compilation property holds as well. Finally, we also discuss the asynchronous synthesis problem. Here, the approach for PLTL and PLDL is similar. Hence, we restrict our attention to the case of PLDL, as it subsumes

PLTL.

5.1. Synchronous Distributed Synthesis for Parametric Linear Temporal Logic

Let $\mathcal{V}$ be an infinite set of variables and let AP be a set of atomic propositions. The formulas of PLTL are given by the grammar

$$\varphi ::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \varphi\,\mathbf{R}\,\varphi \mid \mathbf{F}_{\leq z}\varphi \mid \mathbf{G}_{\leq z}\varphi,$$

where $a \in \mathrm{AP}$ and $z \in \mathcal{V}$. As before, we use the derived operators $\mathbf{F}$ and $\mathbf{G}$ as well as implications, which are defined as for PROMPT–LTL.

The set of sub-formulas of a PLTL formula φ is denoted by $\mathrm{cl}(\varphi)$ and we define the size of φ to be the cardinality of $\mathrm{cl}(\varphi)$. Furthermore, we define

$$\mathrm{var}_{\mathbf{F}}(\varphi) = \{z \in \mathcal{V} \mid \mathbf{F}_{\leq z}\,\psi \in \mathrm{cl}(\varphi)\}$$

to be the set of variables parameterizing eventually operators in φ, and

$$\mathrm{var}_{\mathbf{G}}(\varphi) = \{z \in \mathcal{V} \mid \mathbf{G}_{\leq z}\,\psi \in \mathrm{cl}(\varphi)\}$$

to be the set of variables parameterizing always operators in φ. Finally, $\mathrm{var}(\varphi) = \mathrm{var}_{\mathbf{F}}(\varphi) \cup \mathrm{var}_{\mathbf{G}}(\varphi)$ denotes the set of all variables appearing in φ.

To evaluate formulas, we define a variable valuation to be a mapping $\alpha\colon \mathcal{V} \to \mathbb{N}$ mapping each variable to a value. Now, we can define the model relation between a path $w = w_0w_1w_2\cdots$, a position n of w, a variable valuation α, and a PLTL formula. For the atomic propositions, Boolean connectives, and standard temporal operators, it is defined as for PROMPT–LTL, while for the parameterized operators it is defined as follows:

- $(w, n, \alpha) \models \mathbf{F}_{\leq z}\,\varphi$ if, and only if, there exists a $j \leq \alpha(z)$ such that $(w, n+j, \alpha) \models \varphi$.
- $(w, n, \alpha) \models \mathbf{G}_{\leq z}\,\varphi$ if, and only if, for every $j \leq \alpha(z)$: $(w, n+j, \alpha) \models \varphi$.

For the sake of brevity, we write $(w, \alpha) \models \varphi$ instead of $(w, 0, \alpha) \models \varphi$ and say that w is a model of φ with respect to α.

As usual for parameterized temporal logics, the use of variables has to be restricted: parameterizing eventually and always operators by the same variable leads to an undecidable satisfiability problem [5].

Definition 3. *A* PLTL *formula* φ *is well-formed if* $\mathrm{var}_{\mathbf{F}}(\varphi) \cap \mathrm{var}_{\mathbf{G}}(\varphi) = \emptyset$.

In the following, we only consider well-formed formulas and omit the qualifier "well-formed". Also, we will denote variables in $\mathrm{var}_{\mathbf{F}}(\varphi)$ by x and variables in $\mathrm{var}_{\mathbf{G}}(\varphi)$ by y, if the formula φ is clear from the context.

Our solution for the PLTL synthesis problem is based on the monotonicity of the parameterized temporal operators explained earlier, which is formalized in the following lemma.

Lemma 6 ([5]). *Let φ be a* PLTL *formula and let α and β be variable valuations satisfying $\alpha(x) \leq \beta(x)$, for each $x \in \mathrm{var}_{\mathbf{F}}(\varphi)$, and $\alpha(y) \geq \beta(y)$, for each $y \in \mathrm{var}_{\mathbf{G}}(\varphi)$. If $(w, \alpha) \vDash \varphi$, then $(w, \beta) \vDash \varphi$.*

Thus, let φ be a PLTL formula and let φ' be the PROMPT–LTL-formula obtained from φ by inductively replacing each sub-formula $\mathbf{F}_{\leq x}\,\psi$ by $\mathbf{F_P}\,\psi$ and each sub-formula $\mathbf{G}_{\leq y}\,\psi$ by ψ. The following is a straightforward consequence of the previous lemma.

Corollary 5. *Let φ be a* PLTL *formula and let φ' be defined as above.*

1. *For every w, if there exists a variable valuation α such that $(w, \alpha) \vDash \varphi$, then $(w, \max_{x \in \mathrm{var}_{\mathbf{F}}(\varphi)} \alpha(x)) \vDash \varphi'$.*
2. *For every w, if there exists a bound k such that $(w, k) \vDash \varphi'$, then $(w, \alpha) \vDash \varphi$, where α maps each $x \in \mathrm{var}_{\mathbf{F}}(\varphi)$ to k and each other variable to 0.*

Let $\mathcal{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P}\rangle$ be an architecture. Here, the *synchronous* PLTL *realizability problem for $\mathcal{A}$* is the problem of deciding, given a PLTL formula φ, whether there exist a variable valuation α and a finite-state implementation f_p, for each process $p \in P^-$, such that the distributed product $\bigotimes_{p \in P^-} f_p$ satisfies φ with respect to α, i.e., $(\bigotimes_{p \in P^-} f_p, \alpha) \vDash \varphi$. In this case, we say that φ is realizable in $\mathcal{A}$.

Theorem 4. *Let $\mathcal{A}$ be an architecture. The synchronous* PLTL *realizability problem for $\mathcal{A}$ is decidable if, and only if, $\mathcal{A}$ is weakly ordered.*

Proof. Fix an architecture $\mathcal{A}$. By Corollary 5, a given PLTL formula φ is realizable in $\mathcal{A}$ if, and only if, φ' as defined in the corollary is realizable in $\mathcal{A}$. Thus, Corollary 1 yields the desired result. □

Also, bounded synthesis is again applicable, as we can translate the relativized PLTL formulas into universal co-Büchi automata.

As before, let $\mathcal{V}$ be an infinite set of variables and let AP be the set of atomic propositions. The formulas of PLDL are given by the grammar

$$\begin{aligned}\varphi &::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle g\rangle\, \varphi \mid [g]\, \varphi \mid \langle g\rangle_{\leq z}\, \varphi \mid [g]_{\leq z}\, \varphi \\ g &::= \phi \mid \varphi? \mid g + g \mid g\,;g \mid g^*\end{aligned}$$

where $a \in \mathrm{AP}$, $z \in \mathcal{V}$, and ϕ ranges over propositional formulas over AP. Here, expressions of the form $\varphi?$ are *tests*, which allow us to nest operators. The sets $\mathrm{var}_\Diamond(\varphi)$, $\mathrm{var}_\Box(\varphi)$, and $\mathrm{var}(\varphi)$ are defined analogously to the sets $\mathrm{var}_{\mathbf{F}}(\varphi)$, $\mathrm{var}_{\mathbf{G}}(\varphi)$, and $\mathrm{var}(\varphi)$ for PLTL, taking sub-formulas in tests into account.

The satisfaction relation is defined, as before, among a path w, a position n, a variable valuation α, and a formula φ. First, let the relation $\mathcal{R}(g, w, \alpha) \subseteq \mathbb{N} \times \mathbb{N}$ contain all pairs $(m, n) \in \mathbb{N} \times \mathbb{N}$ such that $w_m \cdots w_{n-1}$ matches g. Formally, it is defined inductively by

- $\mathcal{R}(\phi, w, \alpha) = \{(n, n+1) \mid w_n \vDash \phi\}$ for propositional ϕ,
- $\mathcal{R}(\varphi?, w, \alpha) = \{(n, n) \mid (w, n, \alpha) \vDash \varphi\}$,
- $\mathcal{R}(g_0 + g_1, w, \alpha) = \mathcal{R}(g_0, w, \alpha) \cup \mathcal{R}(g_1, w, \alpha)$,
- $\mathcal{R}(g_0\,; g_1, w, \alpha) = \{(n_0, n_2) \mid \exists n_1 \text{ s.t. } (n_0, n_1) \in \mathcal{R}(g_0, w, \alpha) \text{ and } (n_1, n_2) \in \mathcal{R}(g_1, w, \alpha)\}$, and
- $\mathcal{R}(g^*, w, \alpha) = \{(n, n) \mid n \in \mathbb{N}\} \cup$
  $\{(n_0, n_{k+1}) \mid \exists n_1, \ldots, n_k \text{ s.t. } (n_j, n_{j+1}) \in \mathcal{R}(g, w, \alpha) \text{ for all } j \leq k\}$.

Then, for atomic formulas and Boolean connectives it is defined as for PROMPT–LTL, while for the four temporal operators, it is defined as follows:

- $(w, n, \alpha) \vDash \langle g\rangle\, \varphi$ if there exists $j \geq 0$ such that $(n, n+j) \in \mathcal{R}(g, w, \alpha)$ and $(w, n+j, \alpha) \vDash \varphi$,
- $(w, n, \alpha) \vDash [g]\, \varphi$ if for all $j \geq 0$, with $(n, n+j) \in \mathcal{R}(g, w, \alpha)$, we have $(w, n+j, \alpha) \vDash \varphi$,
- $(w, n, \alpha) \vDash \langle g\rangle_{\leq z}\, \varphi$ if there exists $j \leq \alpha(z)$ such that $(n, n+j) \in \mathcal{R}(g, w, \alpha)$ and $(w, n+j, \alpha) \vDash \varphi$, and

- $(w, n, \alpha) \models [g]_{\leq z} \varphi$ if for all $j \leq \alpha(z)$ with $(n, n+j) \in \mathcal{R}(g, w, \alpha)$, we have $(w, n+j, \alpha) \models \varphi$.

Again, we restrict ourselves to well-formed formulas, i.e., those formulas φ with $\mathrm{var}_{\Diamond}(\varphi) \cap \mathrm{var}_{\Box}(\varphi) = \emptyset$. With this restriction, Lemma 6 holds for PLDL, too.

Lemma 7. *Let φ be a* PLDL *formula and let α and β be variable valuations satisfying $\alpha(x) \leq \beta(x)$, for each $x \in \mathrm{var}_{\Diamond}(\varphi)$, and $\alpha(y) \geq \beta(y)$, for each $y \in \mathrm{var}_{\Box}(\varphi)$. If $(w, \alpha) \models \varphi$, then $(w, \beta) \models \varphi$.*

Recall that the alternating color technique for PROMPT–LTL replaces every prompt-eventually operator $\mathbf{F_P}\, \psi$ by a formula that expresses that ψ holds within one color change. In LTL, this is naturally expressed by two nested until operators. In PLDL, parameterized diamond operators, which are the analogues of prompt-eventually operators, are guarded by regular expressions, and, thus, one has to express that both the guard holds and at most one color change occurs. The simplest way to do it is to introduce a change point bounded variant of the diamond-operator (see [8]).

Formally, we add the operator $\langle \cdot \rangle^{r}_{cp}$ with the following semantics:

- $(w, n, \alpha) \models \langle g \rangle^{r}_{cp} \psi$ if there exists a $j \in \mathbb{N}$ s.t. $(n, n+j) \in \mathcal{R}(g, w, \alpha)$, $w_n \cdots w_{n+j-1}$ contains at most one r-change point, and $(w, n+j, \alpha) \models \psi$.

Let $\mathrm{LDL}_{\mathrm{cp}}$ be the logic obtained by disallowing parameterized operators but allowing the change point-bounded operator, whose semantics are independent of variable valuations. Hence, we drop them from our notation for the satisfaction relation $\models$ and the relation $\mathcal{R}$.

We need the following results from [8] which generalizes the replacement of PLTL sub-formulas $\mathbf{G}_{\leq y}\, \psi$ by ψ with respect to variable valuations mapping y to zero. In PLDL, the situation is different, e.g., the formulas $[g]_{\leq y} \psi$ and ψ are not necessarily equivalent with respect to variable valuations mapping y to zero, e.g., if $r = \varphi?$ is a test. This test has to be satisfied, even if $\alpha(y) = 0$. However, one can easily simplify the guard g to a guard $\widehat{g}$ that captures g when restricted to matchings of length zero.

Lemma 8 ([8])**.** *For every* PLDL *formula φ there is an efficiently constructible* PLDL *formula φ' without paramterized box operators whose size is at most the size of φ such that*

1. $\mathrm{var}_{\Diamond}(\varphi) = \mathrm{var}_{\Diamond}(\varphi')$,

2. *for each α and each w, $(w, \alpha) \models \varphi$ implies $(w, \alpha) \models \varphi'$, and*
3. *for each α and each w, $(w, \alpha) \models \varphi'$ implies $(w, \alpha_0) \models \varphi$.*

In the third item, α_0 is the valuation mapping each $x \in \mathrm{var}_\Diamond(\varphi)$ to $\alpha(x)$ and each other variable to 0.

Note that the formulas φ and φ' as above are equivalent, if the variable valuation is existentially quantified.

Now, given such a PLDL formula φ, let $rel_r(\varphi)$ denote the formula obtained from the formula φ' as in Lemma 8 by inductively replacing each subformula $\langle g \rangle_{\leq x} \psi$ by $\langle g \rangle_{cp}^{r} \psi$. Furthermore, let $alt_r = [\mathbf{tt}^*] \langle \mathbf{tt}^* \rangle r \wedge [true^*] \langle \mathbf{tt}^* \rangle \neg r$, which is equivalent to the LTL formula $\mathbf{GF}\, r \wedge \mathbf{GF} \neg r$ from above. Now, define $c_r(\varphi) = rel_r(\varphi) \wedge alt_r$, which is an $\mathrm{LDL_{cp}}$ formula.

Lemma 9 ([8])**.** *Let φ be a* PLDL *formula and let $w \in \left(2^{\mathrm{AP}}\right)^{\omega}$.*

1. *If $(w, \alpha) \models \varphi$, then $w' \models c_r(\varphi)$ for every k-spaced r-coloring w' of w, where $k = \max_{x \in \mathrm{var}_\Diamond(\varphi)} \alpha(x)$.*
2. *If w' is a k-bounded r-coloring of w with $w' \models c_r(\varphi)$, then $(w, \alpha) \models \varphi$, where α maps each $x \in \mathrm{var}_\Diamond(\varphi)$ to $2k$ and each other variable to zero.*

Finally, the exponential compilation property holds for $\mathrm{LDL_{cp}}$ as well: every $\mathrm{LDL_{cp}}$ formula can be translated into an equivalent non-determinstic Büchi automaton of exponential size [8].

Now, the (synchronous) PLDL distributed synthesis problem is defined as its analogue for PLTL. Let $\mathcal{A} = \langle P, p_{env}, \{I_p\}_{p \in P}, \{O_p\}_{p \in P} \rangle$ be an architecture. Then, the *synchronous* PLDL *realizability problem for $\mathcal{A}$* is the problem of deciding, given a PLDL formula φ, whether there exist a variable valuation α and a finite-state implementation f_p for each process $p \in P^-$, such that the distributed product $\bigotimes_{p \in P^-} f_p$ satisfies φ with respect to α, i.e., $(\bigotimes_{p \in P^-} f_p, \alpha) \models \varphi$. In this case, we say that φ is realizable in $\mathcal{A}$.

Theorem 5. *Let $\mathcal{A}$ be an architecture. The synchronous* PLDL *realizability problem for $\mathcal{A}$ is decidable if, and only if, $\mathcal{A}$ is weakly ordered.*

Proof. Theorem 1 holds for PLDL as well, using the same proof: a PLDL formula φ is realizable in $\mathcal{A}$ if, and only if, $c_r(\varphi)$ is realizable in $\mathcal{A}^r$. Now, the information fork criterion holds for ω-regular conditions as well [13], which finishes the proof. □

Also, bounded synthesis is again applicable, as we can also translate the relativized PLDL formulas into universal co-Büchi automata.

5.3. Asynchronous Distributed Synthesis for PLDL

Finally, we consider the asynchronous setting. We focus on PLDL, as PLTL is a fragment of PLDL and the approach for both problems is similar.

As for the asynchronous PROMPT–LTL realizability problem, we require the implementations to only change their state if they are scheduled. Here, a PLDL assume-guarantee specification $\langle \varphi, \psi \rangle$ consists of a pair of PLDL formulas. The asynchronous PLDL assume-guarantee realizability problem asks, given an asynchronous architecture $\mathcal{A}^*$ and $\langle \varphi, \psi \rangle$ as above, whether there exists a finite-state implementation f_p, for each process $p \in P^-$, such that for each variable valuation α there is a variable valuation β such that for each $w \in \bigotimes_{p \in P^-} f_p$, we have that $(w, \alpha) \vDash \varphi$ implies $(w, \beta) \vDash \psi$. In this case, we say that $\bigotimes_{p \in P^-} f_p$ satisfies $\langle \varphi, \psi \rangle$.

To solve the problem, we use the framework of bounded synthesis and emptiness checking for Büchi graphs as presented for PROMPT–LTL in Section 4. In particular, we adapt the notation introduced in Subsection 4.3, e.g., the product system $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$. Our semi-decision procedure again guesses implementations and then model checks whether their product $\mathcal{S}$ satisfies the assume-guarantee specification, based on a characterization in terms of $\mathcal{S}$ being pumpable non-empty. To this end, we have to lift Lemma 11 to PLDL, which again requires to remove parameterized box operators. Once more, we rely on monotonicity, but due to the quantifier alternation and the implication between φ and ψ, the application is not completely trivial. Given the assumption φ, let φ' be the formula as described in Lemma 8, which has no parameterized box operators. The formula ψ' is defined similarly.

Lemma 10. *Let $\mathcal{S}$, φ', and φ' as above. Then, $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$ if, and only if, $\mathcal{S}$ satisfies $\langle \varphi', \psi' \rangle$.*

Proof. Let f denote the strategy generated by $\mathcal{S}$.

For the implication from left to right, let $\mathcal{S}$ satisfy $\langle \varphi, \psi \rangle$, i.e., for each α there is a β such that for all $w \in f$: $(w, \alpha) \vDash \varphi$ implies $(w, \beta) \vDash \psi$. As β depends on α, we write $\beta(\alpha)$ to make the dependency clear.

Now, given some arbitrary α let α_0 denote the variable valuation mapping each $x \in \mathrm{var}_\Diamond(\varphi) = \mathrm{var}_\Diamond(\varphi')$ to $\alpha(x)$ and each other variable to 0. We claim that $(w, \alpha) \vDash \varphi'$ implies $(w, \beta(\alpha_0)) \vDash \psi'$ for all $w \in f$, which implies that $\mathcal{S}$ satisfies $\langle \varphi', \psi' \rangle$.

Thus, assume the assumption is satisfied, i.e., $(w, \alpha) \vDash \varphi'$. Then, we also have $(w, \alpha_0) \vDash \varphi$ by Lemma 8. Thus, $(w, \beta(\alpha_0)) \vDash \psi$, which implies $(w, \beta(\alpha_0)) \vDash \psi'$, again by Lemma 8.

For the other implication, let $\mathcal{S}$ satisfy $\langle \varphi', \psi' \rangle$, i.e., for each α there is a β such that for all $w \in f$: $(w, \alpha) \models \varphi'$ implies $(w, \beta) \models \psi'$. Again, as β depends on α, we write $\beta(\alpha)$ to make the dependency clear.

We claim that $(w, \alpha) \models \varphi$ implies $(w, \beta(\alpha)) \models \psi$ for all $w \in f$, which implies that $\mathcal{S}$ satisfies $\langle \varphi, \psi \rangle$.

Thus, assume the assumption is satisfied, i.e., $(w, \alpha) \models \varphi$. Then, we also have $(w, \alpha) \models \varphi'$ by Lemma 8. Thus, $(w, \beta(\alpha)) \models \psi'$, which implies $(w, (\beta(\alpha))_0) \models \psi$, again by Lemma 8. Here, $(\beta(\alpha))_0$ maps each variable in $\mathrm{var}_\Diamond(\psi) = \mathrm{var}_\Diamond(\psi')$ to $(\beta(\alpha))(x)$ and each other variable to 0. □

To simplify the notation we can assume that φ and ψ do not contain any parameterized box operators. Thus, the alternating color technique is applicable to them. Also, there is a non-deterministic Büchi automaton $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)} = \langle 2^{I \cup O \cup \{r, r'\}}, Q, q_0, \delta, B \rangle$, where $\overline{c}_{r'}(\psi) = alt_{r'} \wedge \neg rel_{r'}(\psi)$ whose language contains exactly those paths that satisfy $\overline{c}_{r'}(\psi) \wedge c_r(\varphi)$ [8]. Then, Lemma 11 holds in this setting as well.

Lemma 11. *Let $\langle \varphi, \psi \rangle$ be a PLDL assume-guarantee specification, $\mathcal{A}^*$ be an asynchronous architecture, and $\mathcal{S}_p$ be a finite-state implementation for each system process $p \in P^-$. The distributed product $\mathcal{S} = \bigotimes_{p \in P^-} \mathcal{S}_p$ does not satisfy $\langle \varphi, \psi \rangle$ if, and only if, the product of $\mathcal{S}$ and $\mathcal{N}_{\overline{c}_{r'}(\psi) \wedge c_r(\varphi)}$ is pumpable non-empty.*

From here on the algorithm is similar to that described in Section 4 and we obtain the same semi-decidability result.

Corollary 6. *Let $\mathcal{A}^*$ be an asynchronous architecture. The asynchronous PLDL assume-guarantee realizability problem for $\mathcal{A}^*$ is semi-decidable.*

6. Conclusion

In this paper, we have initiated the investigation of distributed synthesis for parameterized specifications, in particular for PROMPT–LTL, PLTL, and PLDL. These logics subsume LTL, and additionally allow to express bounded satisfaction of system properties, instead of only eventual satisfaction. To the best of our knowledge, this is the first treatment of parametrized temporal logic 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. Finally, we have shown that all these results also hold for PLTL and PLDL. Furthermore, the approach is also applicable to PLTL and PLDL in a weighted setting [9], as even these logics have the exponential compilation property and the alternating color technique is applicable to them as well. Finally, we conjecture that the approach also extends to assume-guarantee synthesis with mutual assumptions between different processes [31, 32].

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 [6].

In the future, we also want to look into the synthesis of distributed systems with a parametric number of components [33, 34] from parameterized temporal logics. In addition to the even more difficult problems of realizability and synthesis, new questions arise in this context, such as: how does the bound on prompt eventualities increase with the number of components in the system?

[1] A. Pnueli, The temporal logic of programs, in: FOCS 1977, IEEE, 1977, pp. 46–57. `doi:10.1109/SFCS.1977.32`.

[2] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, Y. Zbar, The ForSpec temporal logic: A new temporal property-specification language, in: J.-P. Katoen, P. Stevens (Eds.), TACAS 2002, Vol. 2280 of LNCS, Springer, 2002, pp. 296–311. `doi:10.1007/3-540-46002-0\_21`.

[3] C. Eisner, D. Fisman, A Practical Introduction to PSL, Integrated Circuits and Systems, Springer, 2006. `doi:10.1007/978-0-387-36123-9`.

[4] M. Y. Vardi, P. Wolper, Reasoning about infinite computations, Inf. Comput. 115 (1) (1994) 1–37. `doi:10.1006/inco.1994.1092`.

[5] R. Alur, K. Etessami, S. La Torre, D. Peled, Parametric temporal logic for "model measuring", ACM Trans. Comput. Log. 2 (3) (2001) 388–407. doi:10.1145/377978.377990.

[6] O. Kupferman, N. Piterman, M. Y. Vardi, From liveness to promptness, Formal Methods in System Design 34 (2) (2009) 83–103. doi:10.1007/s10703-009-0067-z.

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

[8] P. Faymonville, M. Zimmermann, Parametric linear dynamic logic, Inf. Comput. 253 (2017) 237–256. doi:10.1016/j.ic.2016.07.009.

[9] M. Zimmermann, Parameterized linear temporal logics meet costs: still not costlier than LTL, Acta Informatica (2016) 1–24doi:10.1007/s00236-016-0279-9.

[10] A. Pnueli, R. Rosner, Distributed reactive systems are hard to synthesize, in: FOCS 1990, IEEE Computer Society, 1990, pp. 746–757. doi:10.1109/FSCS.1990.89597.

[11] O. Kupferman, M. Y. Vardi, Synthesizing distributed systems, in: LICS 2001, IEEE Computer Society, 2001, pp. 389–398. doi:10.1109/LICS.2001.932514.

[12] S. Mohalik, I. Walukiewicz, Distributed games, in: P. K. Pandya, J. Radhakrishnan (Eds.), FSTTCS 2003, Vol. 2914 of LNCS, Springer, 2003, pp. 338–351. doi:10.1007/978-3-540-24597-1_29.

[13] B. Finkbeiner, S. Schewe, Uniform distributed synthesis, in: LICS 2005, IEEE Computer Society, 2005, pp. 321–330. doi:10.1109/LICS.2005.53.

[14] B. Finkbeiner, S. Schewe, Bounded synthesis, STTT 15 (5-6) (2013) 519–539. doi:10.1007/s10009-012-0228-z.

[15] S. Schewe, B. Finkbeiner, Synthesis of asynchronous systems, in: LOPSTR 2006, Vol. 4407 of LNCS, Springer, 2006, pp. 127–142. doi:10.1007/978-3-540-71410-1_10.

[16] S. Jacobs, L. Tentrup, M. Zimmermann, Distributed PROMPT-LTL synthesis, in: GandALF, Vol. 226 of EPTCS, 2016, pp. 228–241. `doi:10.4204/EPTCS.226.16`.

[17] K. Chatterjee, T. A. Henzinger, J. Otop, A. Pavlogiannis, Distributed synthesis for LTL fragments, in: FMCAD 2013, IEEE, 2013, pp. 18–25. `doi:10.1109/FMCAD.2013.6679386`.

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

[19] B. Finkbeiner, L. Tentrup, Detecting unrealizable specifications of distributed systems, in: Proceedings of TACAS, Vol. 8413 of LNCS, Springer, 2014, pp. 78–92. `doi:10.1007/978-3-642-54862-8_6`.

[20] B. Finkbeiner, L. Tentrup, Detecting unrealizability of distributed fault-tolerant systems, Logical Methods in Computer Science 11 (3). `doi:10.2168/LMCS-11(3:12)2015`.

[21] P. Madhusudan, P. S. Thiagarajan, Distributed controller synthesis for local specifications, in: ICALP 2011, Vol. 2076 of LNCS, Springer, 2001, pp. 396–407. `doi:10.1007/3-540-48224-5\_33`.

[22] W. Fridman, B. Puchala, Distributed synthesis for regular and contextfree specifications, Acta Inf. 51 (3-4) (2014) 221–260. `doi:10.1007/s00236-014-0194-x`.

[23] P. Gastin, N. Sznajder, M. Zeitoun, Distributed synthesis for well-connected architectures, Formal Methods in System Design 34 (3) (2009) 215–237. `doi:10.1007/s10703-008-0064-7`.

[24] P. Gastin, N. Sznajder, Fair synthesis for asynchronous distributed systems, ACM Trans. Comput. Log. 14 (2) (2013) 9. `doi:10.1145/2480759.2480761`.

[25] P. Faymonville, B. Finkbeiner, M. N. Rabe, L. Tentrup, Encodings of bounded synthesis, in: Proceedings of TACAS, Vol. 10205 of LNCS, 2017, pp. 354–370. `doi:10.1007/978-3-662-54577-5_20`.

[26] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.

[27] O. Kupferman, M. Y. Vardi, Safraless decision procedures, in: FOCS, IEEE Computer Society, 2005, pp. 531–542. doi:10.1109/SFCS.2005.66.

[28] M. Leucker, C. Sánchez, Regular linear temporal logic, in: C. Jones, Z. Liu, J. Woodcock (Eds.), ICTAC 2007, Vol. 4711 of LNCS, Springer-Verlag, Macau, China, 2007, pp. 291–305. doi:10.1007/978-3-540-75292-9_20.

[29] P. Wolper, Temporal logic can be more expressive, Information and Control 56 (1–2) (1983) 72 – 99. doi:10.1016/S0019-9958(83)80051-5.

[30] M. Y. Vardi, The rise and fall of LTL, in: G. D'Agostino, S. L. Torre (Eds.), GandALF 2011, Vol. 54 of EPTCS, 2011.

[31] K. Chatterjee, T. A. Henzinger, Assume-guarantee synthesis, in: TACAS, Vol. 4424 of LNCS, Springer, 2007, pp. 261–275. doi:10.1007/978-3-540-71209-1_21.

[32] R. Bloem, K. Chatterjee, S. Jacobs, R. Könighofer, Assume-guarantee synthesis for concurrent reactive programs with partial information, in: TACAS, Vol. 9035 of LNCS, Springer, 2015, pp. 517–532. doi:10.1007/978-3-662-46681-0_50.

[33] S. Jacobs, R. Bloem, Parameterized synthesis, Logical Methods in Computer Science 10 (1). doi:10.2168/LMCS-10(1:12)2014.

[34] A. Khalimov, S. Jacobs, R. Bloem, Towards efficient parameterized synthesis, in: VMCAI, Vol. 7737 of LNCS, Springer, 2013, pp. 108–127. doi:10.1007/978-3-642-35873-9_9.