Specification decomposition for reactive synthesis

Bernd Finkbeiner¹ · Gideon Geier² · Noemi Passing¹

Received: 16 November 2021 / Accepted: 6 June 2022 / Published online: 18 July 2022 © The Author(s) 2022

Abstract

Reactive synthesis is the task of automatically deriving a correct implementation from a specification. It is a promising technique for the development of verified programs and hardware. Despite recent advances in terms of algorithms and tools, however, reactive synthesis is still not practical when the specified systems reach a certain bound in size and complexity. In this paper, we present a sound and complete modular synthesis algorithm that automatically decomposes the specification into smaller subspecifications. For them, independent synthesis tasks are performed, significantly reducing the complexity of the individual tasks. Our decomposition algorithm guarantees that the subspecifications are independent in the sense that completely separate synthesis tasks can be performed for them. Moreover, the composition of the resulting implementations is guaranteed to satisfy the original specification. Our algorithm is a preprocessing technique that can be applied to a wide range of synthesis tools. We evaluate our approach with state-of-the-art synthesis tools on established benchmarks: the runtime decreases significantly when synthesizing implementations modularly.

Keywords Reactive synthesis · Specification decomposition · Modular synthesis · Compositional synthesis · Preprocessing for synthesis

1 Introduction

Reactive synthesis automatically derives an implementation that satisfies a given specification. It is a push-button method producing implementations which are correct by construction. Therefore, reactive synthesis is a promising technique for the development of probably correct systems since it allows for concentrating on what a system should do instead of how it should be done.

Despite recent advances in terms of efficient algorithms and tools, however, reactive synthesis is still not practical when the specified systems reach a certain bound in size and complexity. It is long known that the scalability of model checking algorithms can be improved significantly by using compositional approaches, i.e., by breaking down the analysis of a system into several smaller subtasks [4,6]. In this paper, we apply compositional concepts to reactive synthesis: We present and extend a modular synthesis algorithm [13] that decomposes a specification into several subspecifications. Then, independent synthesis tasks are performed for them. The implementations obtained from the subtasks are combined into an implementation for the initial specification. The algorithm uses synthesis as a black box and can thus be applied on top of a wide range of synthesis algorithms. Thus, it can be seen as a preprocessing step for reactive synthesis that enables compositionality for existing algorithms and tools.

Soundness and completeness of modular synthesis strongly depends on the decomposition of the specification into subspecifications. We introduce a criterion, non-contradictory independent sublanguages, for subspecifications that ensures...
soundness and completeness: the original specification is equirealizable to the subspecifications and the parallel composition of the implementations for the subspecifications is guaranteed to satisfy the original specification. The key question is now how to decompose a specification such that the resulting subspecifications satisfy the criterion.

Lifting the language-based criterion to the automaton level, we present a decomposition algorithm for nondeterministic Büchi automata that directly implements the independent sublanguages paradigm. Thus, using subspecifications obtained with this decomposition algorithm ensures soundness and completeness of modular synthesis. A specification given in the standard temporal logic LTL can be translated into an equivalent nondeterministic Büchi automaton and hence the decomposition algorithm can be applied as well.

However, while the decomposition algorithm is semantically precise, it utilizes several expensive automaton operations. For large specifications, the decomposition thus becomes infeasible. Therefore, we present an approximate decomposition algorithm for LTL specification that still ensures soundness and completeness of modular synthesis but is more scalable. It is approximate in the sense that, in contrast to the automaton decomposition algorithm, it does not necessarily find all possible decompositions. Thus, the decomposition computed by the LTL decomposition algorithm is possibly coarser than the perfect decomposition. Moreover, we present an optimization of the LTL decomposition algorithm for formulas in a common assumption-guarantee format. It analyzes the assumptions and drops those that do not influence the realizability of the rest of the formula, yielding more fine-grained decompositions. We extend the optimization from specifications in a strict assumption-guarantee format to specifications consisting of several conjuncts in assume-guarantee format. This allows for applying the optimization to even more of the common LTL synthesis benchmarks.

We have implemented both decomposition procedures as well as the modular synthesis algorithm and used it with the two state-of-the-art synthesis tools BoSy [10] and Strix [26]. We evaluate our algorithms on the 346 well-established, publicly available benchmarks from the synthesis competition SYNTCOMP [19]. As expected, the decomposition algorithm for nondeterministic Büchi automata becomes infeasible when the specifications grow. For the LTL decomposition algorithm, however, the experimental results are excellent: decomposition terminates in less than 26 ms on all benchmarks. Hence, the overhead of LTL decomposition is negligible, even for non-decomposable specifications. Out of 39 decomposable specifications, BoSy and Strix increase their number of synthesized benchmarks by nine and five, respectively. For instance, on the generalized buffer benchmark [18,21] with three receivers, BoSy is able to synthesize a solution within 28 s using modular synthesis while neither the non-compositional version of BoSy, nor the non-compositional version of Strix terminates within one hour. For twelve and nine further benchmarks, respectively, BoSy and Strix reduce their synthesis times significantly, often by an order of magnitude or more, when using modular synthesis instead of their classical algorithms. The remaining benchmarks are too small and too simple for compositional methods to pay off. Thus, decomposing the specification into smaller subspecifications indeed increases the scalability of synthesis on larger systems.

**Related Work:** Compositional approaches are long known to improve the scalability of model checking algorithms significantly [4,6]. The approach that is most related to our contribution is a preprocessing algorithm for compositional model checking [7]. It analyzes dependencies between the properties that need to be checked in order to reduce the number of model checking tasks. For instance, they search for dependencies of the form \( \varphi_1 \rightarrow \varphi_2 \) which allows them to cancel the model checking task for \( \varphi_2 \) if the one for \( \varphi_1 \) succeeded. We lift the idea of analyzing dependencies in order to improve compositional approaches from model checking to synthesis. However, due to the different nature of compositional model checking and synthesis, the dependency analysis in our approach differs inherently from the one presented in [7] in both their goal and their realization: in compositional model checking, all subtasks consider the same given implementation. Therefore, no conflicts can occur and hence, in [7], the dependency analysis is only used to abort redundant subtasks. In compositional synthesis, in contrast, the solutions of the subtasks define the overall implementation. Therefore, to obtain soundness, we need to ensure that no conflicts in the solutions of the subtasks exist. Thus, our decomposition algorithm aims at identifying subtasks that can be performed individually while guaranteeing that no conflicts in their solutions arise.

There exist several compositional approaches for reactive synthesis. The algorithm by Filiot et al. depends, like our LTL decomposition approach, heavily on dropping assumptions [11]. They use a heuristic that, in contrast to our criterion, is incomplete. Their approach is more scalable than non-compositional synthesis. Yet, one does not see an improvement that is as significant as the one observed for our approach. The algorithm by Kupferman et al. is designed for incrementally adding requirements to a specification during system design [22]. Thus, it does not perform independent synthesis tasks but only reuses parts of the already existing solutions. In contrast to our algorithm, both [22] and [11] do not consider dependencies between the components to obtain prior knowledge about the presence or absence of conflicts in the implementations.

Assume-guarantee synthesis algorithms [2,3,15,24] take dependencies between components into account. In this
setting, specifications are not always satisfiable by one component alone. Thus, a negotiation between the components is needed. While this yields more fine-grained decompositions, it produces a significant overhead that, as our experiments show, is often not necessary for common benchmarks. Avoiding negotiation, dependency-based compositional synthesis [14] decomposes the system based on a dependency analysis of the specification. The analysis is more fine-grained than the one presented in this paper. Moreover, a weaker winning condition for synthesis, remorsefree dominance [5], is used. While this allows for smaller synthesis tasks since the specification can be decomposed further, both the dependency analysis and using a different winning condition produce a larger overhead than our approach.

The reactive synthesis tools Strix [26], Unbeast [9], and Safety-First [32] decompose the given specification. Strix uses decomposition to find suitable automaton types for internal representation and to identify isomorphic parts of the specification. Unbeast and Safety-First in contrast, decompose the specification to identify safety parts. All three tools do not perform independent synthesis tasks for the subspecifications. In fact, our experiments show that the scalability of Strix still improves notably with our algorithm.

Independent of [13], Mavridou et al. introduce a compositional realizability analysis of formulas given in FRET [17] that is based on similar ideas as our LTL decomposition algorithm [25]. They only study the realizability of formulas but do not synthesize solutions. Optimized assumption handling cannot easily be integrated into their approach. For a detailed comparison of both approaches, we refer to [25].

The first version [13] of our modular synthesis approach is already well-accepted in the synthesis community: our LTL decomposition algorithm has been integrated into the new version [31] of the synthesis tool ltlsynt [27].

2 Preliminaries

Notation. Overloading notation, we use union and intersection on infinite words: for \( \sigma = \sigma_1 \sigma_2 \cdots \in (2^{\Sigma_1})^{\omega} \), \( \sigma' = \sigma'_1 \sigma'_2 \cdots \in (2^{\Sigma_2})^{\omega} \) with \( \Sigma = \Sigma_1 \cup \Sigma_2 \), we define \( \sigma \cup \sigma' := (\sigma_1 \cup \sigma'_1) (\sigma_2 \cup \sigma'_2) \cdots \in (2^\Sigma)^{\omega} \). For \( \sigma \) as above and a set \( X \), let \( \sigma \cap X := (\sigma_1 \cap X) (\sigma_2 \cap X) \cdots \in (2^X)^{\omega} \).

LTL. Linear-time temporal logic (LTL) [29] is a specification language for linear-time properties. For a finite set \( \Sigma \) of atomic propositions, the syntax of LTL is given by \( \varphi, \psi ::= a | \text{true} | \neg \varphi | \varphi \lor \psi | \varphi \land \psi | \varphi.U \psi \), where \( a \in \Sigma \). For a trace \( t = t_1 t_2 \cdots \in (2^\Sigma)^{\omega} \), the semantics of an LTL formula is defined by

\[
\begin{align*}
t \models a & \iff a \in t_1 \\
t \models \neg \psi & \iff t \not\models \psi \\
t \models \psi \lor \psi' & \iff t \models \psi \text{ or } t \models \psi' \\
t \models \psi & \iff t[2] \models \psi \\
t \models \psi.U \psi' & \iff \exists i \geq 1. t[i] \models \psi \land \forall j \leq i. t[j] \not\models \psi.
\end{align*}
\]

where \( t[i] \) denotes the infinite subsequence \( t_i t_{i+1} \ldots \) of \( t \) starting from point in time \( i \in \mathbb{N} \). We define the operators \( \Box \varphi := \text{true} . U \varphi \) and \( 
\varphi := \neg \Box \Diamond \neg \varphi \) as usual. The atomic propositions in \( \varphi \) are denoted by \( \text{prop}(\varphi) \), where every occurrence of \( \text{true} \) or \( \text{false} \) in \( \varphi \) does not add any atomic propositions to \( \text{prop}(\varphi) \). The language \( \mathcal{L}(\varphi) \) of \( \varphi \) is the set of infinite words that satisfy \( \varphi \).

Automata. For a finite alphabet \( \Sigma \), a nondeterministic Büchi automaton (NBA) is a tuple \( A = (Q, Q_0, \delta, F) \), where \( Q \) is a finite set of states, \( Q_0 \subseteq Q \) is a set of initial states, \( \delta : Q \times \Sigma \times Q \) is a transition relation, and \( F \subseteq Q \) is a set of accepting states. Given an infinite word \( \sigma = \sigma_1 \sigma_2 \cdots \in \Sigma^{\omega} \) over \( \Sigma \), a run of \( \sigma \) on \( A \) is an infinite sequence \( q_1 q_2 q_3 \cdots \in \Sigma^{\omega} \) of states where \( q_1 \in Q_0 \) and \( (q_i, \sigma_i, q_{i+1}) \in \delta \) holds for all \( i \geq 1 \). A run is accepting if it contains infinitely many accepting states. \( A \) accepts a word \( \sigma \) if there is an accepting run of \( \sigma \) on \( A \). The language \( \mathcal{L}(A) \) of \( A \) is the set of all accepted words. Two NBAs are equivalent if their languages are the same. An LTL specification \( \varphi \) can be translated into an NBA \( A_\varphi \) such that \( \mathcal{L}(\varphi) = \mathcal{L}(A_\varphi) \) with a single exponential blow up [23].

Specifications. A specification \( s \) specifies the behavior of a reactive system. In this paper, we consider specifications given as LTL formulas, also called LTL specifications, as well as specifications given as nondeterministic Büchi automata. When the context is clear, we call both types of specification simply specification.

Implementations and Counterstrategies. An implementation of a system with inputs \( I \), outputs \( O \), and variables \( V = I \cup O \) is a function \( f : (2^V)^* \times 2^I \rightarrow 2^O \) mapping a history of variables and the current input to outputs. An infinite word \( \sigma = \sigma_1 \sigma_2 \cdots \in (2^V)^{\omega} \) over \( 2^V \) is compatible with an implementation \( f \) if for all \( n \in \mathbb{N} \), \( f(\sigma_1 \cdots \sigma_{n-1}, \sigma_n \cap I) = \sigma_n \cap O \) holds. The set of all compatible words of \( f \) is denoted by \( \mathcal{C}(f) \). An implementation \( f \) realizes a specification \( s \) if \( s \in \mathcal{L}(s) \) holds for all \( s \in \mathcal{L}(f) \). A specification is called realizable if there exists an implementation realizing it. If a specification is unrealizable, there is a counterstrategy \( f^c : (2^V)^* \rightarrow 2^I \) mapping a history of variables to inputs. An infinite word \( \sigma = \sigma_1 \sigma_2 \cdots \in (2^V)^{\omega} \) is compatible with \( f^c \) if \( f^c(\sigma_1 \cdots \sigma_{n-1}) = \sigma_n \cap I \) holds for all \( n \in \mathbb{N} \). All compatible words of \( f^c \) violate \( s \), i.e., \( \mathcal{L}(f^c) \subseteq \mathcal{L}(s) \), where \( \mathcal{L}(s) \) denotes the complement of the set \( \mathcal{L}(s) \).

Reactive Synthesis. Given a specification, the realizability problem asks whether there exists an implementation that satisfies the specification for all possible input sequences of the considered reactive system. Synthesis then derives such an implementation whenever one exists. For LTL specification-
Algorithm 1: Modular Synthesis

Input: $s$: Specification, $\text{inp}$: List Var, $\text{out}$: List Var
Output: realizable: Bool, implementation: $\mathcal{T}$
1 subresults $\leftarrow$ decompose($s$, $\text{inp}$, $\text{out}$)
2 sub_results $\leftarrow$ map synthesize_subspecifications
3 foreach (real, strat) $\in$ sub_results do
4   if real then
5      impl $\leftarrow$ extendCounterStrat(strat, $s$)
6 return (impl, strat)
7 impls $\leftarrow$ map secondComponentOfTuple sub_results
8 return ($\mathcal{T}$, compose impls)

3 Modular synthesis

In this section, we introduce a modular synthesis algorithm that divides the synthesis task into independent subtasks by splitting the specification into several subspecifications. The decomposition algorithm has to ensure that the synthesis tasks for the subspecifications can be solved independently and that their results are non-contradictory, i.e., that they can be combined into an implementation satisfying the initial specification. Note that when splitting the specification, we assign a set of relevant in- and output variables to every subspecification. The corresponding synthesis subtask is then performed on these variables.

Algorithm 1 describes this modular synthesis approach. First, the specification is decomposed into a list of subspecifications using an adequate decomposition algorithm (line 1). Then, the synthesis tasks for all subspecifications are solved (line 2). If a subspecification is unrealizable, its counterstrategy is extended to a counterstrategy for the whole specification (line 4 to 6). This construction is given in Definition 3.1. Otherwise, the implementations of the subspecifications (which are the second components of the tuples returned by the synthesis tasks, see line 7) are composed (line 8).

Intuitively, the behavior of the counterstrategy of an unrealizable subspecification $s_i$ violates the full specification $s$ as well. A counterstrategy for the full specification, however, needs to be defined on all variables of $s$, i.e., also on variables that may not occur in $s_i$. Thus, we extend the counterstrategy for $\varphi_i$ such that it ignores outputs outside of $s_i$ and produces an arbitrary valuation of the input variables outside of $s_i$:

Definition 3.1 (Counterstrategy Extension) Let $V_1 \subset V$. Let $s_1$ be an unrealizable subspecification with $\mathcal{L}(s_1) \subseteq (2^V)^\omega$. Let $f_{\mathcal{I}}^{s_1} : (2^V)^{s_1} \rightarrow 2^{|V_1|}$ be a counterstrategy for $s_1$. From $f_{\mathcal{I}}^{s_1}$, we construct a counterstrategy $f^s : (2^V)^s \rightarrow 2^{|V_1|}$ as follows: $f^s(\sigma) = f_{\mathcal{I}}^{s_1}(\sigma \cap V_1) \cup \mu$ for all $\sigma \in (2^V)^s$, where $\mu \in 2^{|V_1|}$ is an arbitrary valuation of the input variables outside of $V_1$.

The counterstrategy for the full specification constructed as in Definition 3.1 then indeed fulfills the condition of a counterstrategy for the full specification, i.e., all of its compatible words violate the full specification, if $s_1$ is a subspecification of $s$ in the sense that all of the requirements posed by $s_1$ are also posed by $s$:

Lemma 3.1 Let $s$ be a specification with $\mathcal{L}(s) \in (2^V)^\omega$. Let $V_1 \subset V$ and let $s_1$ be a subspecification of $s$ with $\mathcal{L}(s_1) \subseteq (2^V)^{s_1}$ and $\{\sigma \cap V_1 \mid \sigma \in \mathcal{L}(s)\} \subseteq \mathcal{L}(s_1)$. Let $s_1$ be unrealizable and let $f_{\mathcal{I}}^{s_1} : (2^V)^{s_1} \rightarrow 2^{|V_1|}$ be a counterstrategy for $s$. The function $f^s$ constructed as in Definition 3.1 from $f_{\mathcal{I}}^{s_1}$ is a counterstrategy for $s$.

Proof Let $\sigma \in \mathcal{C}(f^s)$. Then $f^s(\sigma_1 \ldots \sigma_n-1) = \sigma_n \cap I$ holds for all $n \in \mathbb{N}$ and hence, by construction of $f^s$, we have $f_{\mathcal{I}}^{s_1}(\sigma_1 \ldots \sigma_n-1 \cap V_1) = \sigma_n \cap (I \cap V_1)$. Thus, $\sigma \cap V_1 \in \mathcal{C}(f_{\mathcal{I}}^{s_1})$ follows. Since $f_{\mathcal{I}}^{s_1}$ is a counterstrategy for $s_1$, we have $\mathcal{C}(f_{\mathcal{I}}^{s_1}) \subseteq \overline{\mathcal{L}(s_1)}$. Hence, $\sigma \cap V_1 \in \overline{\mathcal{L}(s_1)}$ holds. By assumption, we have $\{\sigma' \cap V_1 \mid \sigma' \in \mathcal{L}(s)\} \subseteq \mathcal{L}(s_1)$. Thus, in particular, $\sigma \notin \mathcal{L}(s)$ follows. Therefore, for all $\sigma \in \mathcal{C}(f^s)$, $\sigma \notin \mathcal{L}(s)$ and thus $\mathcal{C}(f^s) \subseteq \overline{\mathcal{L}(s)}$. Hence, $f^s$ is a counterstrategy for $s$. 

Soundness and completeness of the modular synthesis algorithm depend on three requirements: (i) equi realizability of the initial specification and the subspecifications, (ii) non-contradictory composability of the subresults, and (iii) satisfaction of the initial specification by the parallel composition of the subresults. Note here that (ii) is a necessary condition for (iii). Intuitively, these requirements are met if the decomposition algorithm neither introduces nor drops parts of the system specification and if it does not produce
subspecifications that allow for contradictory implementations.

We can state all three requirements as requirements on the languages of the subspecifications. To do so, we first define the composition of languages:

**Definition 3.2 (Language Composition)** Let $L_1$, $L_2$ be languages with $L_1 \in (2^{\Sigma_1})^\omega$, $L_2 \in (2^{\Sigma_2})^\omega$, respectively. Their **parallel composition** $L_1 \parallel L_2$ is defined by $L_1\parallel L_2 = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in L_1 \land \sigma_2 \in L_2 \land \sigma_1 \cap \Sigma_2 = \sigma_2 \cap \Sigma_1\}$.

Intuitively, the composition of two languages $L_1$ and $L_2$ is the set of infinite words that combine words in $L_1$ and $L_2$ which agree on shared variables. Thus, in particular, the parallel composition of the languages of two realizable specifications contains only words that agree on both shared input and output variables. To obtain composability of the subresults, all implementations of the specifications need to agree on shared output variables for a given input sequence. This ensures that the implementations do not pose contradictory requirements on output variables. We can thus formulate composability of the subresults in terms of language composition by requiring that the composition of the languages of the subspecifications is again the language of a realizable specification:

**Definition 3.3 (Non-contradictory languages)** Let $V_1$, $V_2 \subseteq V$ with $V_1 \cup V_2 = V$. Let $s_1, s_2$ be specifications with $\emptyset \neq L(s_1) \subseteq (2^{V_1})^\omega$, $\emptyset \neq L(s_2) \subseteq (2^{V_2})^\omega$. If $\forall \gamma \in (2^I)^\omega$. $\exists \sigma \in L(s_1) \parallel L(s_2)$, $\gamma = \sigma \cap I$ holds, then $L(s_1)$ and $L(s_2)$ are called non-contradictory.

Composability of the subresults in modular synthesis then follows from the subspecifications having non-contradictory languages since, intuitively, the subspecifications do not pose contradictory requirements. A more detailed explanation of why composability is ensured follows in the proof of Theorem 3.1.

Note that if two specifications share output variables, their languages can only be non-contradictory if both specifications pose exactly the same requirements on them. For the goal of decomposing specifications to obtain simpler subtasks, requiring replacements in subspecifications is not desirable. Therefore, we only consider subspecifications with non-contradictory languages that do not share output variables. In fact, assigning disjoint sets of output variables to the subtasks of modular synthesis suffices to ensure that the languages of the subspecifications are non-contradictory and thus that the subresults are composable:

**Lemma 3.2** Let $V_1$, $V_2 \subseteq V$. Let $s_1$ and $s_2$ be realizable specifications with $L(s_1) \subseteq (2^{V_1})^\omega$, $L(s_2) \subseteq (2^{V_2})^\omega$. If $V_1 \cap V_2 \subseteq I$ holds, then $L(s_1)$ and $L(s_2)$ are non-contradictory. Moreover, $s_1 \cap s_2 \subseteq L(s_1)$ for all $s_1 \in L(s_1)$, $s_2 \in L(s_2)$ with $s_1 \cap (I \cap V_2) = s_2 \cap (I \cap V_2)$.

**Proof** First, let $s_1 \in L(s_1)$ and $s_2 \in L(s_2)$ be sequences with $s_1 \cap (I \cap V_2) = s_2 \cap (I \cap V_2)$. Since $V_1 \cap V_2 \subseteq I$ holds by assumption, $s_1 \cap V_2 = s_2 \cap V_1$ follows. Hence, $s_1 \cup s_2 \in L(s_1) \parallel L(s_2)$ follows immediately with the definition of language composition.

Next, let $\gamma \in (2^I)^\omega$ be some input sequence. Since $s_1$ and $s_2$ are realizable by assumption, there exist words $s_1 \in L(s_1)$, $s_2 \in L(s_2)$ with $\gamma \cap V_i = s_i \cap I$ for $i \in \{1, 2\}$. Hence, $s_1 \cap (I \cap V_2) = s_2 \cap (I \cap V_2)$ holds. As shown above, $s_1 \cup s_2 \in L(s_1) \parallel L(s_2)$ follows since $V_1 \cap V_2 \subseteq I$. As we chose the input sequence $\gamma$ arbitrarily, it thus follows that $L(s_1)$ and $L(s_2)$ are non-contradictory.

The satisfaction of the initial specification by the composed subresults can be guaranteed by requiring the subspecifications to be independent sublanguages:

**Definition 3.4 (Independent Sublanguages)** Let $L \subseteq (2^\Sigma)^\omega$, $L_1 \subseteq (2^{\Sigma_1})^\omega$, and $L_2 \subseteq (2^{\Sigma_2})^\omega$ be languages with $\Sigma_1, \Sigma_2 \subseteq \Sigma$ and $\Sigma_1 \cup \Sigma_2 = \Sigma$. Then, $L_1$ and $L_2$ are **independent sublanguages of $L$** if $L_1 || L_2 = L$ holds.

From these two requirements, i.e., the subspecifications have non-contradictory languages and they form independent sublanguages, equirealizability of the initial specification and the subspecifications follows:

**Theorem 3.1** Let $s, s_1$, and $s_2$ be specifications with $L(s) \subseteq (2^V)^\omega$, $L(s_1) \subseteq (2^{V_1})^\omega$, and $L(s_2) \subseteq (2^{V_2})^\omega$. If $V_1 \cap V_2 \subseteq I$ and $V_1 \cup V_2 = V$ hold, and $L(s_1)$ and $L(s_2)$ are independent sublanguages of $L(s)$, then $s$ is realizable if, and only if, both $s_1$ and $s_2$ are realizable.

**Proof** First, suppose that $s_1$ and $s_2$ are realizable. Let $f_1 : (2^{V_1})^\ast \times (2^{\cap V_1}) \rightarrow 2^{\cap V_1}$, $f_2 : (2^{V_2})^\ast \times (2^{\cap V_2}) \rightarrow 2^{\cap V_2}$ be implementations realizing $s_1$ and $s_2$, respectively. We construct an implementation $f : (2^V)^\ast \times (2^I)^\ast \rightarrow 2^I$ from $f_1$ and $f_2$ as follows:

$$f(\eta, i) := f_1(\eta \cap V_1, i \cap V_1) \cup f_2(\eta \cap V_2, i \cap V_2).$$

Let $\sigma \in \mathcal{C}(f)$. Hence, $f((\sigma_1 \ldots \sigma_n-1), \sigma_n \cap I) = \sigma_n \cap O$ for all $n \in \mathbb{N}$. Let $\sigma' \in (2^V)^\omega$, $\sigma'' \in (2^V)^\omega$ be sequences with $\sigma'_n \cap O = f_1((\sigma_1 \ldots \sigma_n-1 \cap V_1), \sigma_n \cap (I \cap V_1))$ and $\sigma''_n \cap O = f_2((\sigma_1 \ldots \sigma_n-1 \cap V_2), \sigma_n \cap (I \cap V_2))$, respectively, for all $n \in \mathbb{N}$. Then, $\sigma'$ and $\sigma''$ agree on shared input variables by construction. Hence, since $V_1 \cap V_2 \subseteq V$ and $V_1 \cap V_2 \subseteq V$ holds by assumption, $\sigma' \cup \sigma'' = \sigma$ follows. Moreover, we have $\sigma' \in \mathcal{C}(f_1)$ and $\sigma'' \in \mathcal{C}(f_2)$ and thus, since $s_1$ and $s_2$ are realizable by assumption, $\sigma' \in L(s_1)$ and $\sigma'' \in L(s_2)$ holds. Furthermore, since $V_1 \cap V_2 \subseteq I$ holds, $\sigma'' \cap V_2 = \sigma'' \cap V_2$ follows with Lemma 3.2. Hence, by definition of language composition $\sigma' \cup \sigma'' \in L(s)$ follows. Since $L(s_1)$ and $L(s_2)$ are independent sublanguages by assumption, $L(s_1) \parallel L(s_2) = L(s)$ holds. Thus, $\sigma' \cup \sigma'' \in L(s)$ and hence $\sigma \in \mathcal{C}(s)$. □
Second, let \( s_i \) be unrealizable for some \( i \in \{1, 2\} \) and let \( f^*_i : (2^V)^* \rightarrow 2^I \cap V_i \) be a counterstrategy for \( s_i \). We construct a counterstrategy \( f^* : (2^V)^* \rightarrow 2^I \) from \( f^*_i \) as described in Definition 3.1. Let \( \sigma \in L(s) \). Since we have \( L(s_1) \upharpoonright L(s_2) = L(s) \) by assumption, \( \sigma \in L(s_1) \upharpoonright L(s_2) \) holds as well. In particular, \( \sigma \cap V_i \in L(s_i) \) holds by definition of language composition for \( i \in \{1, 2\} \) and hence we have \( \{ \sigma \cap V_i : \sigma \in L(s) \} \subseteq L(s_i) \). Therefore, it follows with Lemma 3.1 that \( f^* \) is a counterstrategy for \( s \). Thus, \( s \) is unrealizable.

The soundness and completeness of Algorithm 1 for adequate decomposition algorithms now follows directly from Theorem 3.1 and the properties of such algorithms described above: they produce subspecifications that (i) do not share output variables and that (ii) form independent sublanguages of the initial specification.

**Theorem 3.2** Let \( s \) be a specification. Moreover, let \( S = \{s_1, \ldots, s_k\} \) be a set of subspecifications of \( s \) with \( L(s_i) \subseteq (2^V)^\omega \) such that

\[
\bigcup_{1 \leq i \leq k} \bigcup_{i \neq j \leq k} \bigcup_{1 \leq i, j, k} \bigcap_{1 \leq i, j, k} V_i \cap V_j \cap V_k \cap I \subseteq \{ \},
\]

for \( i \neq j \) and such that \( L(s_1), \ldots, L(s_k) \) are independent sublanguages of \( L(s) \). If \( s \) is realizable, Algorithm 1 yields an implementation realizing \( s \). Otherwise, Algorithm 1 yields a counterstrategy for \( s \).

**Proof** First, let \( s \) be realizable. Then, by applying Theorem 3.1 recursively, it follows that \( s_i \) is realizable for all \( i \in S \). Since \( V_i \cap V_j \subseteq I \) holds for any \( s_i, s_j \in S \) with \( i \neq j \), the implementations realizing \( s_1, \ldots, s_k \) are non-contradictory. Hence, Algorithm 1 returns their composition: implementation \( f \). Since \( V_1 \cup \cdots \cup V_k = V \), \( f \) defines the behavior of all outputs. By construction, \( f \) realizes all \( s_j \in S \). Since the \( L(s_i) \) are non-contradictory, independent sublanguages of \( L(s) \), \( f \) thus realizes \( s \).

Next, let \( s \) be unrealizable. Then, by applying Theorem 3.1 recursively, \( s \) is unrealizable for some \( s_j \in S \). Thus, Algorithm 1 returns the extension of \( s_j \)’s counterstrategy to a counterstrategy for the full specification. Its correctness follows, similar to the proof of Theorem 3.1, with Lemma 3.1 and the assumptions posed on the subspecifications and their languages.

**4 Decomposition of Büchi automata**

To ensure soundness and completeness of modular synthesis, a specification decomposition algorithm needs to meet the language-based adequacy conditions of Theorem 3.1. In this section, we lift these conditions from the language level to nondeterministic Büchi automata and present a decomposition algorithm for specifications given as NBAs on this basis. Since the algorithm works directly on NBAs and not on their languages, we consider their composition instead of the composition of their languages: Let \( A_1 = (Q_1, Q_0^1, \delta_1, F) \) and \( A_2 = (Q_2, Q_0^2, \delta_2, F) \) be NBAs with alphabets \( 2^V \), \( 2^V \), respectively. The parallel composition of \( A_1 \) and \( A_2 \) is defined by the NBA \( A_1 \| A_2 = (Q, Q_0, \delta, F) \) with alphabet \( 2^V \times 2^V \) and \( Q = Q_1 \times Q_2, Q_0 = Q_0^1 \times Q_0^2, (q_1, q_2, i, (q'_1, q'_2)) \in \delta \) if, and only if, \( (q_1, i \in V_1, q'_1) \in \delta_1 \) and \( (q_2, i \in V_2, q'_2) \in \delta_2 \) hold, and \( F = F_1 \times F_2 \). The parallel composition of NBAs reflects the composition of their languages:

**Lemma 4.1** Let \( A_1 \) and \( A_2 \) be NBAs with alphabets \( 2^V \) and \( 2^V \). Then, \( L(A_1) \| L(A_2) = L(A_1) \| L(A_2) \) holds.

**Proof** First, let \( \sigma \in L(A_1) \| L(A_2) \). Then, \( \sigma \) is accepted by \( A_1 \| A_2 \). Hence, by definition of automaton composition, for \( i \in \{1, 2\} \), \( \sigma \subseteq V_i \) is accepted by \( A_i \). Thus, \( \sigma \subseteq V_i \subseteq L(A_i) \). Since \( (\sigma \cap V_1) \cap V_2 = (\sigma \cap V_2) \cap V_1 \), we have \( (\sigma \cap V_1) \cup (\sigma \cap V_2) \subseteq L(A_1) \| L(A_2) \). By definition of automaton composition, \( \sigma \subseteq (2^V \times 2^V)\omega \) and thus \( \sigma \subseteq (\sigma \cap V_1) \cup (\sigma \cap V_2) \). Hence, \( \sigma \subseteq L(A_1) \| L(A_2) \).

Next, let \( \sigma \subseteq L(A_1) \| L(A_2) \). Then, for \( \sigma_1 \subseteq (2^V)\omega \) with \( \sigma_1 \subseteq \sigma \), we have \( \sigma_1 \subseteq L(A_1) \) for \( i \in \{1, 2\} \) and \( \sigma_1 \subseteq V_2 = \sigma_2 \cap V_1 \). Hence, \( \sigma_1 \) is accepted by \( A_i \). Thus, by definition of automaton composition and since \( \sigma_1 \subseteq \sigma_2 \) agree on shared variables, \( \sigma_1 \subseteq \sigma_2 \subseteq L(A_1) \| L(A_2) \) and hence \( \sigma \subseteq L(A_1) \| L(A_2) \).
Algorithm 2: Automaton Decomposition

**Name:** decomposeAut

**Input:** $A$: NBA, inp: List Var, out: List Var

**Output:** subautomata: List (NBA, List Var, List Var)

1. if isNull subautomata then
   2. checkedSubsets ← ∅
   3. subautomata ← [(A, inp, out)]
   4. foreach $i \neq X \subseteq \text{out}$ do
      5. $Y \leftarrow \text{out}\setminus X$
      6. if $X \notin \text{checkedSubsets} \land Y \notin \text{checkedSubsets}$ then
         7. $A_X \leftarrow \text{decomposeAut}(A_X, \text{inp, } X)$
         8. $A_Y \leftarrow \text{decomposeAut}(A_Y, \text{inp, } Y)$
         9. if $L(A_X) \cup L(A_Y) \subseteq L(A)$ then
            10. subautomata ← decomposeAut($A_X$, inp, X) ++ decomposeAut($A_Y$, inp, Y)
            11. break
      12. checkedSubsets ← checkedSubsets ∪ [X, Y]
   13. return subautomata

**Example of Specifcation Decomposition Algorithm based onNBAs**

Consider the specification $\phi$ based on NBAs, consider the specification $\phi = \Diamond o_1 \land \Box(i \rightarrow \Diamond o_2)$ for inputs $I = \{i\}$ and outputs $O = \{o_1, o_2\}$. The NBA $A$ that accepts $L(\phi)$ is depicted in Fig. 1. The minimized subautomata obtained with Algorithm 2 are shown in Figs. 1a and 2b. Clearly, $V_1 \cap V_2 \subseteq I$ holds. Moreover, their parallel composition is exactly $A$ depicted in Fig. 1 and therefore their parallel composition accepts exactly those words that satisfy $\phi$. For a slightly modified specification $\phi' = \Diamond o_1 \lor \Box(i \rightarrow \Diamond o_2)$, however, Algorithm 2 does not decompose the NBA $A'$ with $L(A') = L(\phi')$ depicted in Fig. 3: the only possible decomposition is $X = \{o_1\}$, $Y = \{o_2\}$ (or vice-versa), yielding NBAs $A'_X$ and $A'_Y$ that accept every infinite word. Clearly, $L(A'_X) \cup L(A'_Y) \subseteq L(A')$ since $L(A'_X) \cup L(A'_Y) = (2V_1^\omega)^\omega$ and hence $A'_X$ and $A'_Y$ are no valid decomposition.

Algorithm 2 ensures soundness and completeness of modular synthesis: the subspecifications do not share output variables and they are equirealizable to the initial specification. This follows from the construction of the subautomata, Lemma 4.1, and Theorem 3.1:

**Theorem 4.1** Let $A$ be an NBA with alphabet $2^V$. Algorithm 2 terminates on $A$ with a set $S = \{A_1, \ldots, A_k\}$ of NBAs with $L(A_k) \subseteq (2^V_1)^\omega$, where $V_i \cap V_j \subseteq I$ for $1 \leq i, j \leq k$ with $i \neq j$, $V = \bigcup_{1 \leq i \leq k} V_i$, and $A$ is realizable if, and only if, $A_i$ is realizable for all $A_i \in S$.

**Proof** There are NBAs that cannot be decomposed further, e.g., automata whose alphabet contains only one output variable. Thus, since $O$ is finite, Algorithm 2 terminates. We show that the algorithm returns subspecifications that only share input variables, define all output variables of the system, and that are independent sublanguages of the initial specification by structural induction on the initial automaton.

For any automaton $A'$ that is not further decomposable, Algorithm 2 returns a list $S'$ solely containing $A'$. Clearly, the parallel composition of all automata in $S'$ is equivalent to $A'$ and the alphabets of the languages of the subautomata do not share output variables.

Next, let $A'$ be an NBA such that there is a set $X \subseteq \text{out}$ with $L(A'_X) \cup L(A'_Y) \subseteq L(A')$, where $Y = \text{out} \setminus X$. By construction of $A'_X$ and $A'_Y$, we have $(A' \cap (Z \cup \text{inp})) \subseteq A'_X$. For $Z \in \{X, Y\}$. Since both $X \cap Y = \emptyset$ and $X \cup Y = \text{out}$ hold by con-
struction of X and Y, \((X \cup \text{inp}) \cap (Y \cup \text{inp}) \subseteq \text{inp}\) as well as \((X \cup \text{inp}) \cup (Y \cup \text{inp}) = \text{inp} \cup \text{out}\) follows. Therefore, \(\mathcal{L}(A_i) \subseteq \mathcal{L}(A_i') \cup \mathcal{L}(A_i'')\) holds. By induction hypothesis, the recursive calls with \(A_i' \cap \text{inp} \cap \text{out}\) and \(A_i'' \cap \text{inp} \cap \text{out}\) return lists \(S_i'\) and \(S_i''\), respectively, where the parallel composition of all automata in \(S_i'\) is equivalent to \(A_i' \cap \text{inp} \cap \text{out}\) and thus, by construction of \(x\), to \(A_i\). Hence, their languages are independent sublanguages of \(A_i\). Furthermore, for \(Z \in \{X, Y\}\), the alphabets of the automata in \(S_i'\) do not share output variables by induction hypothesis and, by construction, they are subsets of the alphabet of \(A_i' \cap (Z \cup \text{inp})\). Hence, since \((X \cup \text{inp}) \cap (\text{out} \cap \text{out} \cup \text{out}) \subseteq \text{inp}\) holds, the alphabets of the automata in \(S_i' + S_i''\) do not share output variables. Moreover, the union of the alphabets of the automata in \(Z \cap \text{inp} \cap \text{out}\) equals the alphabet of \(A_i' \cap (Z \cup \text{inp})\) for \(Z \in \{X, Y\}\) by induction hypothesis. Since \(X \cup Y = \text{out}\), it follows that the union of the alphabets of the automata in the concatenation of \(S_i'\) and \(S_i''\) equals \(\text{inp} \cup \text{out}\).

Thus, \(\bigcup_{1 \leq i \leq k} V_i = V\) and \(V_i \cap V_j \subseteq I\) for \(1 \leq i, j \leq k\) with \(i \neq j\). Moreover, \(\mathcal{L}(A_1), \ldots, \mathcal{L}(A_k)\) are independent sublanguages of \(\mathcal{L}(A)\). Thus, by Theorem 3.1, \(A\) is realizable if, and only if, all \(A_i \in S\) are realizable. \(\square\)

Since Algorithm 2 is called recursively on every subautomaton obtained by projection, it directly follows that the non-deterministic Büchi automata contained in the returned list are not further decomposable:

**Theorem 4.2** Let \(A\) be an NBA and let \(S\) be the set of NBAs that Algorithm 2 returns on input \(A\). Then, for each \(A_i \in S\) with alphabet \(2^V\), there are no NBAs \(A', A''\) with alphabets \(2^V, 2^V\) with \(V_i = V' \cup V''\) such that \(\mathcal{L}(A_i) = \mathcal{L}(A') \cup \mathcal{L}(A'')\) holds.

Algorithm 2 yields perfect decompositions and is semantically precise. Yet, it performs an exponential number of iterations in the worst case. Moreover, it carries out several expensive automaton operations such as projection, composition, and language containment checks. For the latter, complementation of NBAs is required which is well-known to be problematic in practice. For large automata, Algorithm 2 is thus infeasible. For specifications given as LTL formulas, we therefore present an approximate decomposition algorithm in the next section. While it is not perfect in the sense that the resulting subspecifications may be further decomposable by the automaton decomposition approach, it is free of the expensive automaton operations.

## 5 Decomposition of LTL formulas

An LTL specification can be decomposed by translating it into an equivalent NBA and by then applying Algorithm 2. To circumvent expensive automaton operations, though, we introduce an approximate decomposition algorithm that, in contrast to Algorithm 2, does not necessarily find all possible decompositions. In the following, we assume that \(V = \text{prop}(\varphi)\) holds for the initial specification \(\varphi\). Note that any implementation for the variables in \(\text{prop}(\varphi)\) can easily be extended to one for the variables in \(V\) if \(\text{prop}(\varphi) \subseteq V\) holds by ignoring the inputs in \(I \setminus \text{prop}(\varphi)\) and by choosing arbitrary valuations for the outputs in \(O \setminus \text{prop}(\varphi)\).

The main idea of the decomposition algorithm is to rewrite the initial LTL formula \(\varphi\) into a conjunctive form \(\varphi = \varphi_1 \land \cdots \land \varphi_n\) with as many top-level conjuncts as possible. Then, we build subspecifications \(\varphi_i\) consisting of subsets of the conjuncts. Each conjunct occurs in exactly one subspecification. We say that conjuncts are independent if they do not share output variables. Given an LTL formula with two conjuncts, the languages of the conjuncts are independent sublanguages of the language of the whole formula:

**Lemma 5.1** Let \(\varphi = \varphi_1 \land \varphi_2\) be an LTL formula over atomic propositions \(V\) with conjuncts \(\varphi_1\) and \(\varphi_2\) over \(V_1\) and \(V_2\), respectively, with \(V_1 \cup V_2 \subseteq V\). Then, \(\mathcal{L}(\varphi_1)\) and \(\mathcal{L}(\varphi_2)\) are independent sublanguages of \(\mathcal{L}(\varphi)\).

**Proof** First, let \(\sigma \in \mathcal{L}(\varphi)\). Then, \(\sigma \in \mathcal{L}(\varphi_1)\) holds for all \(i \in \{1, 2\}\). Since \(\text{prop}(\varphi_1) \subseteq V_i\) holds and since the satisfaction of \(\varphi_i\) only depends on the valuations of the variables in \(\text{prop}(\varphi_i)\), we have \(\sigma \cap \varphi_i \in \mathcal{L}(\varphi_i)\). Since clearly \((\sigma \cap V_1) \cap V_2 = (\sigma \cap V_1) \cap V_1\) holds, we have \((\sigma \cap V_1) \cap (\sigma \cap V_2) \subseteq \mathcal{L}(\varphi_1) \cap \mathcal{L}(\varphi_2)\). Since \(V_1 \cup V_2 = V\) holds by assumption, we have \(\sigma = (\sigma \cap V_1) \cup (\sigma \cap V_2)\) and hence \(\sigma \in \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)\) follows.

Next, let \(\sigma \in \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)\). Then, there are words \(\sigma_1 \in \mathcal{L}(\varphi_1), \sigma_2 \in \mathcal{L}(\varphi_2)\) with \(\sigma_1 \cap V_2 = \sigma_2 \cap V_1\) and \(\sigma = \sigma_1 \cup \sigma_2\). Since \(\sigma_1\) and \(\sigma_2\) agree on shared variables, \(\sigma \in \mathcal{L}(\varphi_1)\) and \(\sigma \in \mathcal{L}(\varphi_2)\). Hence, \(\sigma \in \mathcal{L}(\varphi_1 \land \varphi_2)\). \(\square\)

Our decomposition algorithm then ensures that different subspecifications share only input variables, i.e., that the subspecifications are independent, by merging conjuncts that share output variables into the same subspecification. Then, equirealizability of the initial formula and the subformulas follows directly from Theorem 3.1 and Lemma 5.1:

**Corollary 5.1** Let \(\varphi = \varphi_1 \land \varphi_2\) be an LTL formula over \(V\) with conjuncts \(\varphi_1, \varphi_2\) over \(V_1, V_2\), respectively, with \(V_1 \cup V_2 = V\) and \(V_1 \cap V_2 \subseteq I\). Then, \(\varphi\) is realizable if, and only if, both \(\varphi_1\) and \(\varphi_2\) are realizable.

To determine which conjuncts of an LTL formula \(\varphi = \varphi_1 \land \cdots \land \varphi_n\) share variables, we build the dependency graph \(\mathcal{D}_\varphi = (\mathcal{V}, \mathcal{E})\) of the output variables, where \(\mathcal{V} = O\) and \((a, b) \in \mathcal{E}\) if, and only if, \(a \neq b\) and both \(a \in \text{prop}(\varphi_i)\) and \(b \in \text{prop}(\varphi_i)\) for some \(1 \leq i \leq n\). Intuitively, two outputs \(a\) and \(b\) that are contained in the same connected component of \(\mathcal{D}_\varphi\) depend on each other in the sense that they...
Algorithm 3: LTL Decomposition

Input: ψ: LTL, inp: List Var, out: List Var
Output: specs: List (LTL, List Var, List Var)
1 ψ ← rewrite(ψ)
2 formulas ← removeTopLevelConjunction(ψ)
3 graph ← buildDependencyGraph(ψ, out)
4 cc ← graph.connectedComponents()
5 specs ← new LTL[|cc|+1] // init with true
6 foreach ψ ∈ formulas do
7    propositions ← getProps(ψ)
8      foreach (spec, set) ∈ zip specs (cc ++ [inp]) do
9        if propositions ∩ set \neq \emptyset then
10           spec.And(ψ)
11       break
12 return map (λϕ → ((ϕ, inputs(ϕ), outputs(ϕ))) specs

either occur in the same conjunct or that they occur in con-
juncts that are connected by other output variables. Hence,
to ensure that subspecifications do not share outputs, con-
juncts containing a or b need to be assigned to the same
subspecification. Outputs that are contained in different con-
nected components, however, are not linked and therefore
implementations for their requirements can be synthesized
independently, i.e., with independent subspecifications.

Algorithm 3 describes how an LTL formula is decom-
poved into subspecifications. First, the formula is rewritten
into conjunctive form (line 1), e.g., by applying distributiv-
ity and pushing temporal operators inwards whenever possible.
This is done to maximize the number of top-level conjuncts
since the LTL decomposition only decomposes specifications
at conjunctions. Therefore, the formula is decomposed into
its conjuncts (line 2) which serve as potential subspeci-
fications. Then, the dependency graph of the conjuncts is built
(line 3) and its connected components are computed (line
4). For each connected component as well as for all input
variables, a subspecification is built by adding the conjuncts
containing variables of the respective connected component
or an input variable, respectively (line 5 to 11). In more detail,
a list specs of specifications, one for each component and
one for the input variables, is created. Initially, all entries
are true (line 5). Then, in the foreach-loop, the speci-
fications in specs are refined: for each conjunct ψ (line 6),
all specifications are considered (see line 8). If ψ contains
a variable that occurs in the set of variables of the connected
component or input variables (see set) that is assigned to
the currently considered specification spec (line 9), then ψ
is added to spec (line 10).

Note that it is necessary to not only consider the connected
components of the dependency graph but also the input vari-
ables in specs to ensure that every conjunct of the original
specification, including input-only ones, are added to at least
one subspecification. By construction, no conjunct is added to
the subspecifications of two different connected components.
Yet, a conjunct could be added to both a subspecification of a
connected component and the subspecification for the input-
only conjuncts. This is circumvented by the break in line 11.
Hence, every conjunct is added to exactly one subspecification.
To define the input and output variables for the synthesis
subtasks for the subspecifications, the algorithm assigns the
inputs and outputs occurring in ψi to the subspecification
ψi (line 12). While restricting the inputs is not necessary for
correctness, it may improve the runtime of the synthesis task.

As an example for LTL decomposition, consider the spec-
ification ϕ = ◊ ϕ1 ∧ □(i → ϕ2) with I = {i} and
O = {ϕ1, ϕ2}. Since ϕ is already in conjunctive form, no
rewriting is necessart. The two conjuncts of ϕ do not share
any variables and therefore the dependency graph Dϕ does
not contain any edges. Thus, we obtain two subspecifications
ϕ1 = ◊ ϕ1 and ϕ2 = □(i → ϕ2).

Soundness and completeness of modular synthesis with
Algorithm 3 as a decomposition algorithm for LTL formulas
follows directly from Corollary 5.1 if the subspecifications
do not share any output variables:

Theorem 5.1 Let ϕ be an LTL formula over V . Then, Algo-

rithm 3 terminates with a set S = {ϕ1, . . . , ϕk} of LTL
formulas on ϕ with L(ϕi) ∈ (2.V )ω such that V i ∩ V j ⊆ I for
1 ≤ i, j ≤ k with i \neq j, \bigcup_{1 \leq i \leq k} V i = V , and such that ϕ
is realizable, if, and only if, for all subspecifications ϕi ∈ S,
ϕi is realizable.

Proof An output variable is part of exactly one connected
component and all conjuncts containing an output are con-
tained in the same subspecification. Thus, every output is
part of exactly one subspecification and hence V i ∩ V j ⊆ I
holds for all 1 ≤ i \neq j ≤ k. The last component added
in line 8 contains all inputs. Hence, all variables occurring
in ϕ are featured in at least one subspecification. Thus,
\bigcup_{1 \leq i \leq k} V i = prop(ϕ) and hence, since V = prop(ϕ)
by assumption, \bigcup_{1 \leq i \leq k} V i = V follows. Therefore, equireal-
izability of ϕ and the formulas in S directly follows with
Corollary 5.1. □

While Algorithm 3 is simple and ensures soundness and
completeness of modular synthesis, it strongly depends on
the structure of the formula: When rewriting formulas in
assume-guarantee format, i.e., formulas of the form ϕ =
∧_{i=1}^m ϕi → ∧_{j=1}^n ψj, to a conjunctive form, we obtain
ϕ = ∧_{j=1}^n (\bigland_{i=1}^m ϕi → ψj). Hence, if two outputs a and b
occur in assumption ϕi and guarantee ψj, respectively, they
are dependent according to Algorithm 3. Thus, all conjuncts
featuring a or b are contained in the same subspecification
according to Algorithm 3. Yet, ψj might be realizable even
without ϕi. An algorithm accounting for this might yield fur-
ther decompositions and thus smaller synthesis subtasks.

In the following, we present a criterion for dropping
assumptions while maintaining equirealizability. Intuitively,
we can drop an assumption ϕ for a guarantee ψ if they do
not share any variable. However, if ϕ can be violated by the system, i.e., if ¬ϕ is realizable, equiarealizability is not guaranteed when dropping ϕ. For instance, consider the formula ϕ = □(i₁ ∧ o₁) → □(i₂ ∧ o₂), where I = {i₁, i₂} and O = {o₁, o₂}. Although assumption and guarantee do not share any variables, the assumption cannot be dropped: an implementation that never sets o₁ to true satisfies ϕ but □(i₂ ∧ o₂) is not realizable. Furthermore, dependencies between input variables may yield unrealizability if an assumption is dropped as information about the remaining inputs might get lost. For instance, in the formula ϕ with inputs might get lost. For instance, in the formula ϕ = (¬ϕ₁ ∧ ϕ₂) ∧ (¬ϕ₁ ∧ i₂) ∧ (i₃ ↔ ¬i₄) and ψ = i₁ ↔ o, where I = {i₁, i₂, i₃, i₄} and O = {o}, no assumption can be dropped: otherwise the information about the global behavior of i₁, which is crucial for the existence of an implementation, is incomplete. These observations lead to the following criterion for safely dropping assumptions.

Lemma 5.2 Let ψ = (ϕ₁ ∧ ϕ₂) → ψ be an LTL formula with prop(ϕ₁) ∩ prop(ϕ₂) = ∅, prop(ϕ₂) ∩ prop(ψ) = ∅. Let ¬ϕ₂ be unrealizable. Then, ϕ₁ → ψ is realizable if, and only if, ϕ is realizable.

Proof Let V₁ := prop(ϕ₁) ∪ prop(ψ) and V₂ := prop(ϕ₂). For x ∈ {1, 2}, let Iₙ := Vₓ ∩ I and Oₙ := Vₓ ∩ O.

First, suppose that ϕ₁ → ψ is realizable. Then there is an implementation f₁ : (2^V₁)⁰ → 2^O₁ that realizes ϕ₁ → ψ. From f₁, we construct a strategy f : (2^V)⁰ × 2^I → 2^O as follows: let f ∈ 2^V\I be an arbitrary valuation of the outputs outside of O₁. Then, let f(ϕ₁,i) := f₁(ϕ₁ ∧ V₁, i ∩ I₁) ∨ μ. Let σ ∈ C(f). Then we have f(σ₁,0,σₙ₋₁,σₙ ∩ I₁) = σₙ ∩ O for all n ∈ N and thus f₁((σ₁,0,σₙ₋₁,σₙ ∩ I₁, σ ∩ I₁) = σₙ ∩ (O ∩ V₁) follows by construction of f. Since (σ₁ ∩ V₁) ∈ C(f₁) holds and thus, since f₁ realizes ϕ₁ → ψ by assumption, σ ∩ V₁ ∈ C(ϕ₁ → ψ). Since prop(ϕ₁) ∩ prop(ϕ₂) = ∅ and prop(ϕ₂) ∩ prop(ψ) = ∅ hold, we have V₁ ∩ V₂ = ∅. Hence, the valuations of the variables in V₂ do not affect the satisfaction of ϕ₁ → ψ. Thus, we have (σ ∩ V₁) ∪ σ′ ∈ C(ϕ₁ → ψ) for any σ′ ∈ (2^V₂)⁰. In particular, (σ ∩ V₁) ∪ (σ ∩ V₂) ∈ C(ϕ₁ → ψ) holds. Since we have V = prop(ϕ) by assumption, V₁ ∩ V₂ holds. Therefore it follows that (σ ∩ V₁) ∪ (σ ∩ V₂) = σ. Hence, σ ∈ C(ϕ₁ → ψ) implies (ϕ₁ ∧ ϕ₂) → ψ, σ ∈ C(ϕ₁ ∧ ϕ₂) follows. Hence, f realizes ϕ. Next, let (ϕ₁ ∧ ϕ₂) → ψ be realizable. Then, there is an implementation f : (2^V)⁰ × 2^I → 2^O that realizes (ϕ₁ ∧ ϕ₂) → ψ. Since (ϕ₁ ∧ ϕ₂) → ψ is unrealizable by assumption, there is a counterstrategy f₂ : (2^V₂)⁰ → 2^I for ¬ϕ₂. We define a function h : (2^V)⁰ × (2^V)⁰ → (2^V)⁰ that lifts a finite sequence η ∈ (2^V)⁰ from V₁ to V using f and f₂ as follows: for the empty word ε, we define h(τ, ε) = τ. For a finite, nonempty word s · η ∈ (2^V)⁰ over V₁, where s ∈ 2^V and where · : 2^V × (2^V)⁰ → (2^V)⁰ denotes concatenation, we define h(τ, s · η) = h(τ · (s ⊕ I₁) ∪ c ∪ f(τ, (s ⊕ c) ∩ Iₙ), η), where c = f₂(iₙ ∩ V₂). Based on f and h, we construct an implementation g : (2^V₁)⁰ × 2^I → 2^O₁ as well as a function f : (2^V)⁰ × 2^I → 2^V as follows:

g(η, i) := f(h(ε, i), i) ∧ f₂(h(ε, i)) ∧ O₁,

Let σ ∈ C(g) and note that σ ∈ (2^V₁)⁰. Let σ′ ∈ (2^V₂)⁰ be an infinite sequence with f(σ′₁, . . . , σ′ₙ₋₁, σ′ₙ ∩ I₁) = σₙ for all n ∈ N and with σ′ ∩ V₁ = σ. Since f and g agree on the variables O₁ and since f does not alter the variables in I₁, such a sequence σ′ exists. By construction of f, we have σ′ ∈ C(f) and hence, since f realizes ϕ by assumption, σ′ ∈ L(ϕ). Furthermore, σ′ ∩ V₂ ∈ C(f₂) holds by construction of f and since V₁ ∩ V₂ = ∅ by assumption. Since f₂ is a counterstrategy for ¬ϕ₂, all words compatible with f₂ satisfy ϕ₂. Thus, in particular, σ′ ∩ V₂ ∈ L(ϕ₂) holds. Since V₁ ∩ V₂ = ∅, the valuations of the variables in V₁ do not affect the satisfaction of ϕ₂, i.e., (σ′ ∩ V₂) ∪ σ′ ∈ L(ϕ₂) holds for any σ′ ∈ (2^V₂)⁰. Hence, since σ′ ∈ L(ϕ) and since (σ′ ∩ V₂) ∪ σ′ ∈ L(ϕ₂) holds, σ′ ∈ L(ϕ ∧ ϕ₂) holds by definition of ϕ, which is unrealizable. Then, since both σ′ ∈ L(ϕ) and σ′ ∈ L(ϕ₂) hold, we have σ′ ∈ L(ϕ ∧ ϕ₂) and thus, by definition of ϕ, ϕ ∈ L(ϕ₁ → ψ) follows. Since V₁ ∩ V₂ = ∅, the satisfaction of ϕ₁ → ψ is not influenced by the variables outside of V₁. Thus, since we have σ′ ∩ V₁ = σ by construction, σ ∈ L(ϕ₁ → ψ) follows. Hence, g realizes ϕ₁ → ψ. □

By dropping assumptions, we are able to decompose LTL formulas of the form ϕ = ∧₁ⁿ=1 ϕᵢ → ∧₀ˡ=₁ ψᵢ in further cases: we rewrite ϕ to ∧₀ˡ=₁ (∧₁ⁿ=1 ϕᵢ → ψᵢ) and then drop assumptions for the individual guarantees. If the resulting subspecifications only share input variables, they are equiarealizable to ϕ.

Theorem 5.3 Let ϕ = (ϕ₁ ∧ ϕ₂ ∧ ϕ₃) → (ψ₁ ∧ ψ₂) be an LTL formula over V, where prop(ϕₙ) ⊆ I and prop(ϕ₁) ∩ prop(ϕ₂) ⊆ I. Let prop(ϕ₁) ∩ prop(ϕᵢ) = ∅ for i, j ∈ {1, 2, 3} with i ≠ j, and prop(ϕᵢ) ∩ prop(ϕ₃₋₁) = ∅ for i ∈ {1, 2}. Let (ϕ₁ ∧ ϕ₂ ∧ ϕ₃) be unrealizable. Then, ϕ is realizable if, and only if, both ϕ′ = (ϕ₁ ∧ ϕ₂) → ψ₁ and ϕ″ = (ϕ₂ ∧ ϕ₃) → ψ₂ are realizable.

Proof Define Vᵢ := prop(ϕᵢ) ∪ prop(ϕ₃) ∪ prop(ϕ₃) for i ∈ {1, 2, 3}. Since we have V = prop(ϕ) by assumption, V₁ ∩ V₂ = V holds. With the assumptions made on ϕ₁, ϕ₂, ϕ₃, ψ₁, and ψ₂, we obtain V₁ ∩ V₂ ⊆ I.

First, let ϕ be realizable and let f : (2^V)⁰ × 2^I → 2^O be an implementation that realizes ϕ. Let σ ∈ C(f). Then,
\[ \sigma \in \mathcal{L}(\varphi) \text{ and thus by the semantics of implication, } \sigma \cap (V \setminus \text{prop}(\varphi_{3-i})) \in \mathcal{L}((\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_i) \] follows for \( i \in \{1, 2\} \). Hence, an implementation \( f_i \) that behaves as \( f \) restricted to \( O \setminus \text{prop}(\varphi_{3-i}) \) realizes \((\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_i\). By Lemma 5.2, \((\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_i \) and \((\varphi_1 \wedge \varphi_3) \rightarrow \psi_i \) are equirealizable since \( \varphi_1, \varphi_2, \) and \( \varphi_3 \) as well as \( \varphi_{3-i} \) and \( \psi_i \) do not share any variables. Thus, there exist implementations \( f_1 \) and \( f_2 \) realizing \((\varphi_1 \wedge \varphi_3) \rightarrow \psi_1 \) and \((\varphi_2 \wedge \varphi_3) \rightarrow \psi_2 \), respectively.

Next, let both \((\varphi_1 \wedge \varphi_3) \rightarrow \psi_1 \) and \((\varphi_2 \wedge \varphi_3) \rightarrow \psi_2 \) be realizable and let \( f_1 : (2^{|V|})^* \times 2^{|\bigcap V_1|} \rightarrow 2^{|O \cap V_1|} \) be an implementation realizing \((\varphi_1 \wedge \varphi_3) \rightarrow \psi_1 \). We construct an implementation \( f : (2^{|V|})^* \times 2^{|V_1|} \rightarrow 2^{|O \cap V_1|} \) from \( f_1 \) and \( f_2 \) as follows: \( f(\eta, i) := f_1(\eta \cap V_1, i \cap V_1) \cup f_2(\eta \cap V_2, i \cap V_2) \). Let \( \sigma \in \mathcal{C}(f) \). Since \( V_1 \) and \( V_2 \) do not share any output variables, \( \sigma \cap V_1 \in \mathcal{L}((\varphi_1 \wedge \varphi_3) \rightarrow \psi_1) \) follows from the construction of \( f \). Moreover, \( \sigma \cap V_1 \) and \( \sigma \cap V_2 \) agree on shared variables and thus \((\sigma \cap V_1) \cup (\sigma \cap V_2) \in \mathcal{L}(\varphi_1 \wedge \varphi_3)\). Therefore, we have \((\sigma \cap V_1) \cup (\sigma \cap V_2) \in \mathcal{L}(\varphi)\) as well by the semantics of conjunction and implication. Since \( V_1 \cup V_2 = V \), we have \((\sigma \cap V_1) \cup (\sigma \cap V_2) = \sigma \) and thus \( \sigma \in \mathcal{L}(\sigma) \). Hence, \( f \) realizes \( \varphi \).

Analyzing assumptions thus allows for decomposing LTL formulas in further cases and still ensures soundness and completeness of modular synthesis. In the following, we present an optimized LTL decomposition algorithm that incorporates assumption dropping into the search for independent conjuncts. Note that the algorithm is only applicable to LTL formulas that are given in strict assume-guarantee format, i.e., formulas of the form \( \varphi = \bigwedge_{i=1}^{m} \psi_i \rightarrow \bigwedge_{j=1}^{n} \psi_j \). Intuitively, the algorithm needs to identify variables that cannot be shared safely among subspecifications. If an assumption contains such non-shareable variables, we say that it is bound to guarantees since it can influence the possible decompositions. Otherwise, it is called free.

To determine which assumptions are relevant for decomposition, i.e., which assumptions are bounded assumptions, we build a slightly modified version of the dependency graph that is only based on assumptions and not on all conjuncts of the formula. Moreover, all variables serve as the nodes of the graph, not only the output variables. An undirected edge between two variables in the modified dependency graph denotes that the variables occur in the same assumption. Variables that are contained in the same connected component as an output variable \( o \in O \) are thus connected to \( o \) over a path of one or more assumptions. Therefore, they may not be shared among subspecifications as they might influence \( o \) and thus may influence the decomposability of the specification. These variables are then called decomposition-critical. Note that, since the modified dependency graph contains all variables as nodes and not only the ones that are contained in assumptions, all output variables of the system are by construction decomposition-critical. Given the modified dependency graph, we can compute the decomposition-critical propositions with a simple depth-first search.

In Fig. 5, an example for determining decomposition-critical variables using the modified dependency graph (see Fig. 4a) is given. Since the modified dependency graph is built solely from the assumptions, only \( \varphi \) is relevant. Since \( i_1 \) and \( o_1 \) are contained in the same connected component and since \( o_2 \) is an output variable, \( i_1 \) is decomposition-critical. All output variables \( o_1, o_2, \) and \( o_3 \) are decomposition-critical as well. Input \( i_2 \), in contrast, is not decomposition-critical.

After computing the decomposition-critical propositions, we create the dependency graph and extract connected components in the same way as in Algorithm 3 to decompose the LTL specification. Instead of using only output variables as nodes of the graph, though, we use all decomposition-critical variables. Consider the example from Fig. 5 again. In Fig. 4b, the dependency graph based on the decomposition-critical variables is depicted. Since \( i_2 \) is not decomposition-critical, it is not contained in the graph. Since \( o_2 \) and \( o_3 \) are contained in the same connected component, the guarantee conjuncts \( o_2 \wedge i_2 \), \( i_1 \rightarrow \neg o_3 \), and \( o_3 \) are dependent. Note that this is indeed necessary for equirealizability. In contrast, \( i_2 \rightarrow o_1 \) does not depend on the other guarantee conjuncts and vice versa since they only share the non-critical input \( i_2 \).

The LTL decomposition algorithm with optimized assumption handling is shown in Algorithm 4. After building both dependency graphs (line 3 and 4), we identify free assumptions (line 11) and add all other assumptions to their subspecifications similar to Algorithm 3 (line 13 to 16). The guarantees are assigned to their subspecifications in the same manner (line 17 to 22). Lastly, we add the free assumptions to the subspecifications (line 23). Since they are free, they can be safely added to all subspecifications. To obtain small subspecifications, however, we only add them to subspecifications for which they are needed: those featuring variables that occur in the assumption.
The decomposition algorithm does not check for assumption violations. The unrealizability of the negation of the dropped assumption, however, is an essential part of the criterion for assumption dropping (c.f. Theorem 5.2). Therefore, we incorporate the check for assumption violations into the modular synthesis algorithm: before decomposing the specification, we perform synthesis on the negated assumptions. If synthesis returns that the negated assumptions are realizable, the system is able to violate an assumption. The implementation satisfying the negated assumptions is then extended to an implementation for the whole specification that violates the specifications consisting of several assume-guarantee conjuncts. In the next section, we thus present an extension of the LTL decomposition algorithm with optimized assumption handling to specifications consisting of several assume-guarantee conjuncts, i.e., specifications of the form $\varphi = (\varphi_1 \rightarrow \psi_1) \land \cdots \land (\varphi_k \rightarrow \psi_k)$.

6 Optimized LTL decomposition for formulas with several assume-guarantee conjuncts

Since Corollary 5.1 can be applied recursively, classical LTL decomposition, i.e., as described in Algorithm 3, is applicable to specifications with several conjuncts. That is, in particular, it is applicable to specifications with several assume-guarantee conjuncts, i.e., specifications of the form $\varphi \equiv (\varphi_1 \rightarrow \psi_1) \land \cdots \land (\varphi_k \rightarrow \psi_k)$. Algorithm 4, in contrast, is restricted to LTL specifications consisting of a single assume-guarantee pair since Theorem 5.2, on which Algorithm 4 relies, assumes a top-level implication in the specification. Hence, we cannot apply the optimized assumption handling to specifications with several assume-guarantee conjuncts directly.

A naive approach to extend assumption dropping to formulas with several assume-guarantee conjuncts is to first drop assumptions for all conjuncts separately and then to decompose the resulting specification using Algorithm 3. In general, however, this is not sound: the other conjuncts may introduce dependencies between assumptions and guarantees that prevent the dropping of the assumption. When considering the conjuncts during the assumption dropping phase separately, however, such dependencies are not detected. For instance, consider a system with $I = \{i\}$, $O = \{o_1, o_2\}$, and the specification $\varphi = \Box \neg(o_1 \land o_2) \land \Box (i \leftrightarrow o_1) \land (\Box i \rightarrow \Box o_2)$. Clearly, $\varphi$ is realizable by an implementation that sets $o_1$ at $\neg i$ and $o_2$ at $i$ in every time step. Since the first conjunct contains both $o_1$ and $o_2$, Corollary 5.1 is not applicable and thus Algorithm 3 does not decompose $\varphi$. The naive approach for incorporating assumption dropping described above considers the third conjunct of $\varphi$ separately and checks whether the assumption $\Box i$ can be dropped. Since the assumptions and guarantees do not share any variables, Lemma 5.2 is applicable and thus the naive algorithm drops $\Box i$, yielding $\varphi' = \Box \neg(o_1 \land o_2) \land \Box (i \leftrightarrow o_1) \land \Box o_2$. Yet, $\varphi'$ is not realizable: if $i$ is constantly set to $false$, the second con-
junct of ϕ′ enforces o1 to be always set to true. The third conjunct enforces o2 to be constantly set to true irrespective of the input i. The first conjunct, however, requires in every time step one of the output variables to be false. Thus, although Lemma 5.2 is applicable to \( \square i \rightarrow \square o_1 \), dropping the assumption safety is not possible in the context of the other two conjuncts. In particular, the first conjunct of ϕ introduces a dependency between o1 and o2 while the second conjunct introduces one between i and o1. Hence, there is a transitive dependency between i and o1 due to which the assumption \( \square i \) cannot be dropped. This dependency is not detected when considering the conjuncts separately during the assumption dropping phase.

In this section, we introduce an optimization of the LTL decomposition algorithm which is able to decompose specifications with several conjuncts (possibly) in assumption-guarantee format and which is, in contrast to the naive approach described before, sound. Similar to the naive approach, the main idea is to first check for assumptions that can be dropped in the different conjuncts and to then perform the classical LTL decomposition algorithm. Yet, the assumption dropping phase is not performed completely separately for the individual conjuncts but takes the other conjuncts and thus possible transitive dependencies between the assumptions and guarantees into account.

If the other conjuncts do not share any variable with the assumption to be dropped, then there are no transitive dependencies between the assumption and the guarantee due to the other conjuncts. Thus, the assumption can be dropped safely if the other conditions of Lemma 5.2 are satisfied:

**Lemma 6.1** Let \( ϕ = ϕ_1 \land ((ϕ_1 \land ϕ_2) \rightarrow ϕ_2) \) be an LTL formula, where we have \( prop(ϕ_1) \land prop(ϕ_2) = \emptyset \), \( prop(ϕ_1) \land prop(ϕ_2) = \emptyset \) and \( prop(ϕ_1) \land prop(ϕ_2) = \emptyset \). Let \( \neg ϕ_2 \) be unrealizable. Then, \( ϕ’ = ϕ_1 \land (ϕ_1 \rightarrow ϕ_2) \) is realizable if, and only if, \( ϕ \) is realizable.

**Proof** Let \( V_i := \text{prop}(ϕ_1 \land (ϕ_1 \rightarrow ϕ_2)) \), \( V_2 := \text{prop}(ϕ_2) \). For \( i \in \{1, 2\} \), let \( I_i := V_i \cap I \) and \( O_i := V_i \cap O \). First, suppose that \( \neg ϕ_2 \) is realizable. Then, we can construct an implementation \( f : (2^V)^* \times 2^I \rightarrow 2^O \) that realizes \( ϕ’ \) from the implementation \( f_1 : (2^V)^* \times 2^I \rightarrow 2^O \) that realizes \( ϕ’ \) analogous to the proof of Lemma 5.2.

Next, let \( ϕ \) be realizable. Then, there is an implementation \( f : (2^V)^* \times 2^I \rightarrow 2^O \) that realizes \( ϕ \). Since \( \neg ϕ_2 \) is unrealizable by assumption, there is a counterstrategy \( f_2^c : (2^V)^* \rightarrow 2^E \) for \( \neg ϕ_2 \). All words compatible with \( f_2^c \) satisfy \( ϕ_2 \). Let \( g : (2^V)^* \times 2^I \rightarrow 2^O \) and \( f : (2^V)^* \times 2^I \rightarrow 2^V \) be the implementation and the function constructed from \( f \) and \( f_2^c \) in the proof of Lemma 5.2. In the following, we show that \( g \) realizes \( ϕ’ \). Let \( σ' \in L(ϕ) \). Then, \( σ’ \in (2^V)^n \) be an infinite sequence with \( f(o_1' \ldots o_{n-1}' \land o_n' \land h) = σ_n' \) for all \( n \in \mathbb{N} \) and with \( σ' \cap V_1 = σ \). As shown in the proof of Lemma 5.2, both \( σ' \in L(ϕ) \) and \( σ' \in L(ϕ_2) \) hold. Thus, \( σ' \in L(ϕ_1 \land (ϕ_1 \rightarrow ϕ_2)) \) follows. Since \( ϕ_2 \) neither shares variables with \( ϕ_1 \) nor with \( ϕ_1 \lor ϕ_2 \), the satisfaction of \( ϕ_1 \land (ϕ_1 \rightarrow ϕ_2) \) is not influenced by the variables outside of \( V_1 \). Hence, since \( σ' \cap V_1 = σ \) by construction, \( σ \in L(ϕ') \) follows and thus \( g \) realizes \( ϕ' \).

Similar to the optimized assumption handling for specifications in strict assume-guarantee form described in the previous section, we utilize Lemma 6.1 for an optimized decomposition for specifications containing several assume-guarantee conjuncts: We rewrite LTL formulas of the form \( ϕ = ϕ_1 \land \bigwedge_{i=1}^n ϕ_i \rightarrow \bigwedge_{i=1}^n ϕ_j \rightarrow ϕ' \land \bigwedge_{i=1}^n (ϕ_i \rightarrow ϕ_j) \) and then drop assumptions for the individual guarantees \( ϕ_1, \ldots, ϕ_j \) according to Lemma 6.1. If the resulting subspecifications only share input variables, they are equirealizable to \( ϕ \).

**Theorem 6.1** Let \( ϕ = ϕ_1 \land ϕ_2 \land (ϕ_1 \land ϕ_2 \land ϕ_3) \rightarrow ϕ_1 \land ϕ_2 \) be an LTL formula over \( V \), where \( prop(ϕ_1) \subseteq I \) and \( prop(ϕ_2) \subseteq O \). Let \( ϕ \) be unrealizable. Then, there is an implementation \( f \) from \( \square i \rightarrow \square o_1 \) to be always set to false of the input \( t \) to be unrealizable. Then, \( ϕ \) is realizable if, and only if, both \( ϕ' = ϕ_1 \land (ϕ_1 \land ϕ_3) \) and \( ϕ'' = ϕ_2 \land (ϕ_2 \land ϕ_3) \) are realizable.

**Proof** First, let \( ϕ' \) and \( ϕ'' \) be realizable. Then, there are implementations \( f_1 \) and \( f_2 \) realizing \( ϕ' \) and \( ϕ'' \), respectively. Since \( ϕ' \) and \( ϕ'' \) do not share output variables by assumption, we can construct an implementation realizing \( ϕ \) from \( f_1 \) and \( f_2 \) as in the proof of Theorem 5.2.

Next, \( ϕ \) be realizable and let \( f : (2^V)^* \times 2^I \rightarrow 2^O \) be an implementation realizing \( ϕ \). Let \( σ \in L(ϕ) \). Then, \( σ \in L(ϕ) \) holds. Let \( V' = prop(ϕ_1) \cup prop(ϕ_2) \) and let \( V'' = prop(ϕ_1) \cup prop(ϕ_2) \). Then, \( σ \in L(ϕ) \) holds. Let \( σ \cap V' = L(ϕ_1) \) and \( σ \cap V'' = L(ϕ_2) \) as well as \( σ \cap V'' = L(ϕ_2) \) follow. Thus, an implementation \( f_1 \) that behaves as \( f \) restricted to the variables in \( V' \) realizes \( ϕ_1 \land (ϕ_1 \land ϕ_2 \rightarrow ϕ_3) \). An implementation \( f_2 \) that behaves as \( f \) restricted to the variables in \( V'' \) realizes \( ϕ_1 \land (ϕ_1 \land ϕ_2 \rightarrow ϕ_3) \) and \( ϕ' \). By assumption, for \( i \in \{1, 2\}, ϕ_i \) does not share any variables with \( ϕ_3, ϕ_2-1, ϕ_3-1 \) and \( ϕ_3-1 \). Therefore, by Lemma 6.1, \( ϕ' \land (ϕ_1 \land ϕ_2 \land ϕ_3) \rightarrow ϕ_1 \) and \( ϕ' \) are equirealizable. Moreover, \( ϕ'' \land (ϕ_1 \land ϕ_2 \land ϕ_3) \rightarrow ϕ_2 \) and \( ϕ'' \) are equirealizable. Thus, since \( f_1 \) and \( f_2 \) realize the former formulas, \( ϕ' \) and \( ϕ'' \) are both realizable.

Utilizing Theorem 6.1, we extend Algorithm 4 to LTL specifications that do not follow a strict assume-guarantee form but consist of multiple conjuncts. The extended algorithm is depicted in Algorithm 6. We assume that the specification is not decomposable by Algorithm 3, i.e., we assume that no plain decompositions are possible. In practice,
we thus first rewrite the specification and apply Algorithm 3 before then applying Algorithm 6 to the resulting specifications.

Hence, we assume that the dependency graph built from the output propositions of all given conjuncts consists of a single connected component. Theorem 6.1 hands us the tools to “break a link” in that chain of dependencies. This link has to be induced by a suitable implication. Algorithm 6 assumes that at least one of the conjuncts is an implication. In case of several implications, the choice of the implication consequently determines whether or not a decomposition is found. Therefore, it is crucial to reapply the algorithm on the subspecifications after a decomposition has been found and to try all implications if no decomposition is found. Since iterating through all conjuncts does not pose a large overhead in computing time, the choice of the implication is not further specified in the algorithm.

The extended algorithm is similar to Algorithm 4. Note that the dependency graph used for finding the decomposition-critical propositions (see line 4) is built only from the assumptions of the chosen implication as we are only seeking for droppable assumptions of this implication. The dependency graph for identifying independent subspecifications (see line 5), in contrast, includes the dependencies induced by all conjuncts, not only the ones induced by the chosen implication. As in Algorithm 4, the graph is built over all decomposition-critical variables and not only over output variables to ensure that assumptions of the chosen implication may only be dropped if they do not share any variables with the remaining conjuncts. An example for both types of dependency graphs is given in Fig. 5. Intuitively, the remaining conjuncts are thus treated in the same way as the guarantees of the chosen implication. This carries over to when the conjuncts are added to the subspecifications (line 24 to 29). Lastly, Algorithm 6 slightly differs from Algorithm 4 when adding the free assumptions to the subspecifications (line 30). Here, the remaining conjuncts have to be considered, too, since we may not drop assumptions that share variables with the outside conjunct. Consequently, all free assumptions that share an input with one of the remaining conjuncts, needs to be added.

One detail that has to be taken into account when integrating this LTL decomposition algorithm with extended optimized assumption handling into a synthesis tool, is that, like Algorithm 4, Algorithm 6 assumes that all negated assumptions are unrealizable. For formulas in a strict assume-guarantee format, the consequences of realizable assumptions is that we have found a strategy for the implementation. This changes when considering formulas with additional conjuncts since they might forbid this strategy. To detect such strategies, we can verify the synthesized strategy against the remaining conjunct and only extend it to a counterstrategy for the whole specification in the positive case.

7 Experimental evaluation

We implemented the modular synthesis algorithm as well as the decomposition approaches and evaluated them on the 346 publicly available SYNTCOMP [19] 2020 benchmarks. Note that only 207 of the benchmarks have more than one output variable and are therefore realistic candidates for
decomposition. The automaton decomposition algorithm utilizes Spot’s [8] automaton library (Version 2.9.6). The LTL decomposition relies on SyFCo [20] for formula transformations (Version 1.2.1.1). We first decompose the specification with our algorithms and then run synthesis on the resulting subspecifications. Note that our setting, strategies can be seen as circuits with latches (representing memories of the circuit from the previous time step) and gates (representing the control logic of the current time step). We compare the CPU time of the synthesis task as well as the number of gates, and latches of the synthesized circuit (in AIGER format [1]) for the original specification to the sum of the corresponding attributes of all subspecifications. Note that parallelization of the synthesis tasks may further reduce the runtime.

7.1 LTL decomposition

The LTL decomposition algorithm with optimized assumption handling (c.f. Sect. 6) terminates on all benchmarks in less than 26 milliseconds. Thus, even for non-decomposable specifications, the overhead of trying to perform decompositions first is negligible. Algorithm 6 decomposes 39 formulas into several subspecifications; most of them yielding two or three subspecifications. Only a handful of formulas are decomposed into more than six subspecifications. One of the decomposed specifications is unrealizable, all others are realizable. The full distribution of the number of resulting subspecifications for all specifications is shown in Table 1. The basic LTL decomposition approach (see Algorithm 3) yields only 24 decompositions. Therefore, assumption dropping has a considerable impact on the performance of the decomposition algorithm.

We evaluate our modular synthesis approach with two state-of-the-art synthesis tools: BoSy [10], a bounded synthesis tool, and Strix [26], a game-based synthesis tool, both in their 2019 release. We used a machine with a 3.6GHz quad-core Intel Xeon processor and 32GB RAM as well as a timeout of 60 min.

In Fig. 6, the comparison of the runtimes for non-compositional synthesis and modular synthesis are shown for the decomposable SYNTCOMP benchmarks. Note that due to the negligible runtime of specification decomposition, the plot looks similar when considering all SYNTCOMP benchmarks instead. The plot relates the accumulated runtime of all benchmarks solved so far (y-axis) to the number of solved instances (x-axis). Thus, the graph that is most right when reaching the timeout (here, this is modular synthesis with Strix) solved the most instances during the given 60 min. For both BoSy and Strix, one can observe that decomposition generates a slight overhead for small specifications. For larger and more complex specifications, however, modular synthesis decreases the execution time significantly, often by an order of magnitude or more.

Table 2 shows the running times of BoSy and Strix for modular and non-compositional synthesis on exemplary benchmarks. For modular synthesis, the accumulated running time of all synthesis tasks is depicted. On almost all of them, both tools decrease their synthesis times with modular synthesis notably compared to the original non-compositional approaches. Particularly noteworthy is the benchmark generalized_buffer_3. In the last synthesis competition, SYNTCOMP 2021, no tool was able to synthesize a solution for it within one hour. With modular synthesis, however, BoSy yields a result in less than 28 s.

In Tables 3 and 4, the number of gates and latches, respectively, of the AIGER circuits [1] corresponding to the implementations computed by BoSy and Strix for modular and non-compositional synthesis are depicted for exemplary benchmarks. For most specifications, the solutions of modular synthesis are of the same size or smaller in terms of gates than the solutions for the original specification. The size of the solutions in terms of latches, however, varies. Note that BoSy does not generate solutions with less than one latch in general. Hence, the modular solution will always have at least as many latches as subspecifications.
Table 2 Synthesis time in seconds of BoSy and Strix for non-compositional and modular synthesis on exemplary SYNTCOMP benchmarks with a timeout of 60 min

| Benchmark         | Original |         | Modular  |         |         |         | # subspec. |
|-------------------|----------|---------|----------|---------|---------|---------|------------|
|                   | BoSy     | Strix   | BoSy     | Strix   |         |         |            |
| Cockpitboard      | 1526.32  | 11.06   | **2.108**| 8.168   | 8       |         |            |
| Gamelogic         | TO       | 1062.27 | TO       | **25.292**| 4       |         |            |
| LedMatrix         | TO       | TO      | TO       | **1156.68**| 3       |         |            |
| Radarboard        | TO       | 126.808 | 1.54     | 2.744   | 2       |         |            |
| Zoo10             | 1.316    | 1.54    | **0.884**| 11.04   | 11      |         |            |
| generalized_buffer_2 | 70.71  | 534.732 | **4.188**| 7.892   | 2       |         |            |
| generalized_buffer_3 | TO     | TO      | TO       | **27.136**| 3       | 319.988 |            |
| shift_8           | **0.404**| 1.336   | 2.168    | 3.6     | 8       |         |            |
| shift_10          | **1.172**| 1.896   | 2.692    | 4.464   | 10      |         |            |
| shift_12          | 4.336    | 6.232   | **3.244**| 5.428   | 12      |         |            |

The fastest synthesis time is marked in bold font.

Table 3 Gates of the synthesized solutions of BoSy and Strix for non-compositional and modular synthesis on exemplary SYNTCOMP benchmarks

| Benchmark         | Original |         | Modular  |         |         |         |          |
|-------------------|----------|---------|----------|---------|---------|---------|----------|
|                   | BoSy     | Strix   | BoSy     | Strix   |         |         |          |
| Cockpitboard      | 11       | 7       | 25       | 10      |         |         |          |
| Gamelogic         | –        | 26      | –        | **21**  |         |         |          |
| LedMatrix         | –        | –       | –        | **97**  |         |         |          |
| Radarboard        | –        | 6       | 19       | 6       |         |         |          |
| Zoo10             | 14       | 15      | 15       | **13**  |         |         |          |
| generalized_buffer_2 | 3       | 12      | 3        | 11      |         |         |          |
| generalized_buffer_3 | –       | –       | **20**   | 3772    |         |         |          |
| shift_8           | 8        | 0       | 8        | 7       |         |         |          |
| shift_10          | 10       | 0       | 10       | 9       |         |         |          |
| shift_12          | 12       | 0       | 12       | 11      |         |         |          |

Entry – denotes that no solution was found within 60 min
The smallest number of gates is marked in bold font.

Table 4 Latches of the synthesized solutions of BoSy and Strix for non-compositional and modular synthesis on exemplary SYNTCOMP benchmarks

| Benchmark         | Original |         | Modular  |         |         |         |          |
|-------------------|----------|---------|----------|---------|---------|---------|----------|
|                   | BoSy     | Strix   | BoSy     | Strix   |         |         |          |
| Cockpitboard      | 1        | 0       | 8        | 0       |         |         |          |
| Gamelogic         | –        | 2       | –        | **2**   |         |         |          |
| LedMatrix         | –        | –       | –        | **5**   |         |         |          |
| Radarboard        | –        | 0       | 11       | 0       |         |         |          |
| Zoo10             | 1        | 2       | 2        | 2       |         |         |          |
| generalized_buffer_2 | 69     | 47134   | **14**   | 557     |         |         |          |
| generalized_buffer_3 | –       | –       | **3**    | 14      |         |         |          |
| shift_8           | 1        | 0       | 8        | 0       |         |         |          |
| shift_10          | 1        | 0       | 10       | 0       |         |         |          |
| shift_12          | 1        | 0       | 12       | 0       |         |         |          |

Entry – denotes that no solution was found within 60 min
The smallest number of latches is marked in bold font.

7.2 Automaton decomposition

Besides LTL specifications, Strix also accepts specifications given as deterministic parity automata (DPAs) in extended HOA format [28], an automaton format well-suited for synthesis. Thus, our implementation for decomposing specifications given as NBAs performs Algorithm 2, converts the resulting automata to DPAs and then synthesizes solutions with Strix.

For 235 out of the 346 benchmarks, NBA decomposition terminates within 10 min, yielding several subspecifications or proving that the specification is not decomposable. The distribution of the number of subspecifications for all specifications is shown in Table 5. In 79 of the other cases, the tool timed out after 60 min and in the remaining 32 cases it reached the memory limit of 16GB or the internal limits of Spot. Note, however, that for 81 of these specifications even plain DPA generation failed.

Analyzing the instances on which automaton decomposition did not find a solution within 60 min reveals that for these benchmarks the NBAs for the initial, non-decomposed specifications are already very large in terms of states (more than 1000) or in terms of both states (more than 100) and atomic propositions (more than 20). Thus, automaton decomposition becomes infeasible when the specifications grow.

Yet, when comparing the distribution of number of subspecifications for the LTL approach (c.f., Table 1) and the NBA approach (c.f., Table 5), it becomes clear that the automaton decompositions yields more fine-grained decompositions. Thus, it allows for smaller and hence potentially easier synthesis subtasks. However, the coarser LTL decomposition suffices to reduce the synthesis time on common benchmarks significantly. Thus, LTL decomposition is in the right balance between small subtasks and a scalable decomposition.
indicating that modular synthesis is particularly beneficial in system designs, for instance the Syntroids [16] case study, exceptionally well for specifications that stem from real able specifications. We observed that decomposition works (LTL) and 36.75% (NBA) of the potentially decomposable positions found by the LTL approach, the automaton decomposition algorithm times out or reaches the memory limit. However, with unlimited resources the automaton approach should find the same (or finer) decompositions as it is able to find perfect decompositions.

Recall that only 207 of the 346 SYNTCOMP benchmarks are realistic candidates for specification decomposition. For those, our automaton decomposition algorithm proves that 90 of these specifications (43.6%) are not decomposable. Thus, our implementations yield decompositions for 33.33% (LTL) and 36.75% (NBA) of the potentially decomposable specifications. We observed that decomposition works exceptionally well for specifications that stem from real system designs, for instance the Syntroids [16] case study, indicating that modular synthesis is particularly beneficial in practice.

8 Conclusion

We have presented a modular synthesis algorithm that applies compositional techniques to reactive synthesis. It reduces the complexity of synthesis by decomposing the specification in a preprocessing step and then performing independent synthesis tasks for the subspecifications. We have introduced a criterion for decomposition algorithms that ensures soundness and completeness of modular synthesis as well as two algorithms for specification decomposition satisfying the criterion: a semantically precise one for specifications given as nondeterministic Büchi automata, and an approximate algorithm for LTL specifications. We presented optimizations of the LTL decomposition algorithm for formulas in a strict assume-guarantee format and for formulas consisting of several assume-guarantee conjuncts. Both optimizations are based on dropping assumptions that do not influence the realizability of the rest of the formula. We have implemented the modular synthesis algorithm as well as both decomposition algorithms and we compared our approach for the state-of-the-art synthesis tools BoSy and Strix to their non-compositional forms. Our experiments clearly demonstrate the significant advantage of modular synthesis with LTL decomposition over traditional synthesis algorithms. While the overhead is negligible, both BoSy and Strix are able to synthesize solutions for more benchmarks with modular synthesis than in their non-compositional form. Moreover, on large and complex specifications, BoSy and Strix improve their synthesis times notably, demonstrating that specification decomposition is a game-changer for practical LTL synthesis.

Building up on the presented approach, we can additionally analyze whether the subspecifications fall into fragments for which efficient synthesis algorithms exist, for instance safety specifications. Since modular synthesis performs independent synthesis tasks for the subspecifications, we can choose, for each synthesis task, an algorithm that is tailored to the fragment the respective subspecification lies in. Moreover, parallelizing the individual synthesis tasks may increase the advantage of modular synthesis over classical algorithms. Since the number of subspecifications computed by the LTL decomposition algorithm highly depends on the rewriting of the initial formula, a further promising next step is to develop more sophisticated rewriting algorithms.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Biere A, Heljanko K, Wieringa S (2011) AIGER 1.9 And Beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria
2. Bloem R, Chatterjee K, Jacobs S, Königshofer R (2015) Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier C, Tinelli C (eds) Tools and algorithms for the construction and analysis of systems—21st international conference, TACAS 2015. Held as Part of the European joint conferences on theory and practice of software, ETAPS 2015. Proceedings, Lecture notes in computer science, vol 9035. Springer, Berlin, pp 517–532. https://doi.org/10.1007/978-3-662-46681-0_50

3. Chatterjee K, Henzinger TA (2007) Assume-guarantee synthesis. In: Grumberg O, Huth M (eds) Tools and algorithms for the construction and analysis of systems, 13th international conference, TACAS 2007, held as part of the joint European conferences on theory and practice of software, ETAPS 2007. Proceedings, Lecture notes in computer science, vol 4424. Springer, Berlin, pp 261–275. https://doi.org/10.1007/978-3-540-71209-1_21

4. Clarke EM, Long DE, McMillan KL (1989) Compositional model checking. In: Proceedings of the fourth annual symposium on logic in computer science, LICS 1989. IEEE Computer Society, pp 353–362. https://doi.org/10.1109/LICS.1989.39190

5. Damm W, Finkbeiner B (2011) Does it pay to extend the perimeter of a world model? In: Butler MJ, Schulte W (eds) Formal methods—17th international symposium on formal methods, FM 2011. Proceedings, lecture notes in computer science, vol 6664. Springer, Berlin, pp 12–26. https://doi.org/10.1007/978-3-642-21437-0_4

6. de Roever WP, Langmaack H, Pnueli A (eds) (1998) Compositionality: the significant difference, COMPOS 1997, Lecture notes in computer science, vol 1536. Springer, Berlin. https://doi.org/10.1007/1-58113-4652-0_3

7. Dureja R, Rozier KY (2018) More scalable LTL model checking via discovering design-space dependencies (D 3). In: Beyer D, Huisman M (eds) Tools and algorithms for the construction and analysis of systems—24th international conference, TACAS 2018, held as part of the European joint conferences on theory and practice of software, ETAPS 2018. Proceedings, Part I, Lecture notes in computer science, vol 10805. Springer, Berlin, pp 309–327. https://doi.org/10.1007/978-3-319-89960-2_17

8. Duret-Lutz A, Lewkowicz A, Fauchille A, Michaud T, Renault E, Xu L (2016) Spot 2.0—a framework for LTL and ω-automata manipulation. In: Artho C, Legay A, Peled D (eds) Automated technology for verification and analysis—14th international symposium, ATVA 2016. Proceedings, Lecture notes in computer science, vol 9938, pp 122–129. https://doi.org/10.1007/978-3-319-46520-3_8

9. Ehlers R (2011) Unbeast: Symbolic bounded synthesis. In: Tools and algorithms for the construction and analysis of systems—17th international conference, TACAS 2011, held as part of the joint European conferences on theory and practice of software, ETAPS 2011. Proceedings, Lecture notes in computer science, vol 6605. Springer, Berlin, pp 272–275. https://doi.org/10.1007/978-3-642-19835-9_25

10. Faymonville P, Finkbeiner B, Tentrup L (2017) BoSy: an experimentation framework for bounded synthesis. In: Majumdar R, Kunčak V (eds) Computer aided verification—29th international conference, CAV 2017. Proceedings, Part II, Lecture notes in computer science, vol 10427. Springer, Berlin, pp 325–332. https://doi.org/10.1007/978-3-319-63396-9_17

11. Filieri E, Jin N, Raskin J (2010) Compositional algorithms for LTL synthesis. In: Bouajjani A, Chin W (eds) Automated technology for verification and analysis—8th international symposium, ATVA 2010. Proceedings, Lecture notes in computer science, vol 6252. Springer, Berlin, pp 112–127. https://doi.org/10.1007/978-3-642-15643-4_10

12. Finkbeiner B (2016) Synthesis of reactive systems. In: Esparza J, Grumberg O, Sickert S (eds) Dependable software systems engineering, NATO science for peace and security series-D: information and communication security, vol 45. IOS Press, pp 72–98. https://doi.org/10.3233/978-1-61499-627-9_72

13. Finkbeiner B, Geier G, Passing N (2021) Specification decomposition for reactive synthesis. In: NASA Formal Methods, NFM 2021. Proceedings

14. Finkbeiner B, Passing N (2020) Dependency-based compositional synthesis. In: Hung DV, Sokolsky O (eds) Automated technology for verification and analysis—18th international symposium, ATVA 2020. Proceedings, Lecture notes in computer science, vol 12302. Springer, Berlin, pp 447–463. https://doi.org/10.1007/978-3-030-59152-6_25

15. Finkbeiner B, Passing N (2021) Compositional synthesis of modular systems. In: Hou Z, Ganesh V (eds) Automated technology for verification and analysis—19th international symposium, ATVA 2021. Gold Coast, QLD, Australia, October 18–22, 2021. Proceedings, Lecture notes in computer science, vol 12971. Springer, Berlin, pp 303–319. https://doi.org/10.1007/978-3-030-88885-5_20

16. Geier G, Heim P, Klein F, Finkbeiner B (2019) Synontrois: synthesizing a game for fpgas using temporal logic specifications. In: Barrett CW, Yang J (eds) 2019 formal methods in computer aided design, FMCAD 2019. Proceedings. IEEE, pp 138–146. https://doi.org/10.23919/FMCAD.2019.8894261

17. Giannakopoulou T, Pressburger T, Mavridou A, Rhein J, Schumann J, Shi N (2020) Formal requirements elicitation with FRET. In: Sabztaieh M, Vogelsang A, Abualhaija S, Borg M, Dalpiaz F, Daneva M, Condori-Fernández N, Franch X, Fucci D, Gervasi V, Groen EC, Guizzardi RSS, Herrmann A, Horkoff J, Mich L, Perini A, Susi A (eds) Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020. CEUR Workshop Proceedings, vol 2584. CEUR-WS.org. http://ceur-ws.org/V ol-2584/PT-paper4-pdf

18. Jacobs S, Bloem R (2016) The reactive synthesis competition: SYNTCOMP 2016 and beyond. In: Piskac R, Dimitrova R (eds) Fifth workshop on synthesis, SYNT@CA V 2016. Proceedings, EPTCS, vol 229, pp 133–148. https://doi.org/10.4204/EPTCS.229.11

19. Jacobs S, Bloem R, Colange M, Faymonville P, Finkbeiner B, Khalimov A, Klein F, Luttenberger M, Meyer PJ, Michaud T, Sakr M, Sickert S, Tentrup L, Walker A (2019) The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, Participants & Results. CoRR arXiv:1904.07736

20. Jacobs S, Klein F, Schirmer S (2016) A high-level LTL synthesis format: TLSF v1.1. In: Piskac R, Dimitrova R (eds) Fifth workshop on synthesis, SYNT@CA V 2016. Proceedings, EPTCS, vol 229, pp 112–132. https://doi.org/10.4204/EPTCS.229.10

21. Jobstmann B (2007) Applications and optimizations for LTL synthesis. Ph.D. thesis, Graz University of Technology

22. Kupferman O, Piterman N, Vardi MY (2006) Safraless compositional synthesis. In: Ball T, Jones RB (eds) Computer aided verification, 18th international conference, CAV 2006. Proceedings, Lecture notes in computer science, vol 4144. Springer, Berlin, pp 31–44. https://doi.org/10.1007/11817963_6

23. Kupferman O, Vardi MY (2005) Safraless decision procedures. In: 46th annual IEEE symposium on foundations of computer science (FOCS). Proceedings. IEEE Computer Society, pp 531–542 (2005)

24. Majumdar R, Mallik K, Schmuck A, Zufferey D (2020) Assume-guarantee distributed synthesis. IEEE Trans Comput Aided Des Integr Circuits Syst 39(11):3215–3226. https://doi.org/10.1109/ TCAD.2020.3012641

25. Mavridou A, Katis A, Giannakopoulou D, Kooi D, Pressburger T, Whalen MW (2021) From partial to global assume-guarantee contracts: compositional realizability analysis in FRET. In: Huisman
M. Pasareanu CS, Zhan N (eds) Formal methods—24th international symposium, FM 2021, virtual event, November 20–26, 2021, Proceedings, Lecture notes in computer science, vol 13047. Springer, Berlin, pp 503–523 (2021). https://doi.org/10.1007/978-3-030-90870-6_27

26. Meyer PJ, Sickert S, Luttenberger M (2018) Strix: explicit reactive synthesis strikes back! In: Chockler H, Weissenbacher G (eds) Computer aided verification—30th international conference, CAV 2018, held as part of the federated logic conference, FloC 2018. Proceedings, Part I, Lecture notes in computer science, vol 10981. Springer, Berlin, pp 578–586 (2018). https://doi.org/10.1007/978-3-319-96145-3_31

27. Michaud T, Colange M (2018) Reactive synthesis from LTL specification with Spot. In: 7th workshop on synthesis, SYNT@CAV. https://www.lrde.epita.fr/dload/papers/michaud.18.synt.pdf

28. Pérez GA (2019) The extended HOA format for synthesis. CoRR arXiv:1912.05793

29. Pnueli A (1977) The temporal logic of programs. In: Annual symposium on foundations of computer science. IEEE Computer Society, pp 46–57

30. Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: Conference record of the sixteenth annual ACM symposium on principles of programming languages. ACM Press, pp 179–190. https://doi.org/10.1145/75277.75293

31. Renkin F, Duret-Lutz A, Schlehuber P, Pommellet A (2021) Improvements to litsynth. In: 10th workshop on synthesis, SYNT@CAV. https://www.lrde.epita.fr/~frenkin/publications/syntcomp21.pdf

32. Sohail S, Somenzi F (2013) Safety first: a two-stage algorithm for the synthesis of reactive systems. Int J Softw Tools Technol Transf 15(5–6):433–454. https://doi.org/10.1007/s10009-012-0224-3

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.