Specification Decomposition for Reactive Synthesis

Bernd Finkbeiner[1], Gideon Geier[2], and Noemi Passing[1](✉)

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner,noemi.passing}@cispa.de

[2] Saarland University, Saarbrücken, Germany
geier@react.uni-saarland.de

Abstract. Reactive synthesis is the task of automatically deriving an implementation from a specification. It is a promising technique for the development of verified programs and hardware. Despite recent advances, reactive synthesis is still not practical when the specified systems reach a certain bound in size and complexity. In this paper, we present a modular synthesis algorithm that decomposes the specification into smaller subspecifications. For them, independent synthesis tasks are performed, and the composition of the resulting implementations is guaranteed to satisfy the full 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 and obtain encouraging results: The overall runtime decreases significantly when synthesizing implementations modularly.

1 Introduction

Reactive synthesis automatically derives an implementation that satisfies a given specification. Thus, it is a promising technique for the development of provably correct systems. Despite recent advances, however, reactive synthesis is still not practical when the specified systems reach a certain bound in size and complexity. In verification, breaking down the analysis of a system into several smaller subtasks has proven to be a key technique to improve scalability [4,26]. In this paper, we apply compositional concepts to reactive synthesis.

We present a modular synthesis algorithm 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. Since the algorithm uses synthesis

An extended version of this paper is available at [12].

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

A. Dutle et al. (Eds.): NFM 2021, LNCS 12673, pp. 113–130, 2021.
https://doi.org/10.1007/978-3-030-76384-8_8

as a black box, it can be applied to a wide range of synthesis algorithms. In particular, the algorithm can be seen as a preprocessing step for synthesis.

Soundness and completeness of modular synthesis depends on the decomposition. We introduce a criterion, *non-contradictory independent sublanguages*, for subspecifications that ensures soundness and completeness. The key question is now how to decompose a specification such that the criterion is satisfied.

Lifting the language-based criterion to an automaton level, we propose a decomposition algorithm for specifications given as nondeterministic Büchi automata that directly implements the independent sublanguages paradigm. Thus, using subspecifications obtained with this 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 algorithm is semantically precise, it involves several expensive automaton operations. Thus, for large specifications, the decomposition becomes infeasible. Therefore, we present an approximate decomposition algorithm for LTL formulas that still ensures soundness and completeness of modular synthesis but is more scalable. It is approximate in the sense that it does not necessarily find all possible decompositions. Besides, we introduce an optimization of this algorithm for formulas in a common assumption-guarantee format.

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 [9] and Strix [22]. We evaluated our algorithms on the set of established benchmarks from the synthesis competition SYNTCOMP [16]. 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 26ms on all benchmarks, and hence the overhead is negligible. 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 [15,18] with three receivers, BoSy is able to synthesize a solution within 28 s using modular synthesis while neither of the non-compositional approaches terminates within one hour. For twelve and nine further benchmarks, respectively, BoSy and Strix reduce the synthesis times significantly with modular synthesis, often by an order of magnitude or more. 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: In model checking, compositional approaches improve the scalability of algorithms significantly [26]. The approach that is most related to our contribution is a preprocessing algorithm for model checking [6]. It analyzes dependencies between the properties to be checked to reduce the number of model checking tasks. We lift this idea from model checking to reactive synthesis. Our approach, however, differs inherently in the dependency analysis.

There exist several compositional synthesis approaches. The algorithm by Kupferman et al. is designed for incrementally adding requirements to a specification during system design [19]. Thus, it does not perform independent synthesis tasks but only reuses parts of the already existing solutions. The algorithm by Filiot et al. depends, like our LTL decomposition approach, heavily on dropping assumptions [10]. They use an heuristic that, in contrast to our criterion, is incomplete. While their approach is more scalable than a non-compositional one, one does not see as significant differences as for our algorithm. Both algorithms do not consider dependencies between the components to obtain prior knowledge about the presence or absence of conflicts in the implementations.

Assume-guarantee synthesis [2,3,21] takes 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 an enormous overhead that, as our experiments show, is often not necessary for common benchmarks. Avoiding negotiation, dependency-based compositional synthesis [13] 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, it also produces a larger overhead than our approach.

The synthesis tools Strix [22], Unbeast [8], and Safety-First [27] decompose the specification. The first one does so to find suitable automaton types for internal representation and to identify isomorphic parts, while the last two identify safety parts. They do not perform independent synthesis tasks for the subspecifications. In fact, the scalability of Strix improves notably with our algorithm.

2 Preliminaries

LTL. Linear-time temporal logic (LTL) [24] is a specification language for linear-time properties. Let Σ be a finite set of atomic propositions and let $a \in \Sigma$. The syntax of LTL is given by $\varphi, \psi ::= a \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \psi$. We define $\mathit{true} := a \vee \neg a$, $\mathit{false} := \neg\mathit{true}$, $\Diamond\varphi := \mathit{true}\,\mathcal{U}\,\varphi$, and $\Box\varphi := \neg\Diamond\neg\varphi$ and use standard semantics. The atomic propositions in φ are denoted by $\mathit{prop}(\varphi)$, where every occurrence of *true* or *false* in φ does not add any atomic propositions to $\mathit{prop}(\varphi)$. The language $\mathcal{L}(\varphi)$ of φ is the set of infinite words that satisfy φ.

Automata. For a finite alphabet Σ, a nondeterministic Büchi automaton (NBA) is a tuple $\mathcal{A} = (Q, Q_0, \delta, F)$, 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 $\sigma = \sigma_1\sigma_2 \cdots \in \Sigma^\omega$, a run of σ on $\mathcal{A}$ is an infinite sequence $q_1q_2q_3 \cdots \in Q^\omega$ 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 called accepting if it contains infinitely many visits to accepting states. $\mathcal{A}$ accepts a word σ if there is an accepting run of σ on $\mathcal{A}$.

The language $\mathcal{L}(\mathcal{A})$ of an NBA $\mathcal{A}$ is the set of all accepted words. Two NBAs are equivalent if their languages are equivalent. An LTL specification φ can be translated into an equivalent NBA $\mathcal{A}_\varphi$ with a single exponential blow up [20].

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 $\sigma = \sigma_1\sigma_2 \cdots \in (2^V)^\omega$ is compatible with an implementation f if for all $n \in \mathbb{N}$, $f(\sigma_1 \ldots \sigma_{n-1}, \sigma_n \cap I) = \sigma_n \cap O$ holds. The set of all compatible words of f is denoted by $\mathcal{C}(f)$. An implementation f realizes a specification s if $\sigma \in \mathcal{L}(s)$ holds for all $\sigma \in \mathcal{C}(f)$. 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 $f^c(\sigma_1 \ldots \sigma_{n-1}) = \sigma_n \cap I$ holds for all $n \in \mathbb{N}$. All compatible words of f^c violate s, i.e., $\mathcal{C}(f^c) \subseteq \overline{\mathcal{L}(s)}$.

Reactive Synthesis. Given a specification, reactive synthesis derives an implementation that realizes it. For LTL specifications, synthesis is 2EXPTIME-complete [25]. Since we use synthesis as a black box procedure in this paper, we do not go into detail here. Instead, we refer the interested reader to [11].

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

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. Then, the synthesis tasks for all subspecifications are solved. If a subspecification is unrealizable, its counterstrategy is extended to a counterstrategy for the whole specification. This construction is given in the full version [12]. Otherwise, the implementations of the subspecifications are combined.

Soundness and completeness of modular synthesis depend on three requirements: Equirealizability of the initial specification and the subspecifications, non-contradictory composability of the subresults, and satisfaction of the initial specification by the parallel composition of the subresults. Intuitively, these

Algorithm 1: Modular Synthesis

```
   Input: s: Specification, inp: List Variable, out: List Variable
   Output: realizable: Bool, implementation: $\mathcal{T}$
 1 subspecifications ← decompose(s, inp, out)
 2 sub_results ← map synthesize subspecifications
 3 foreach (real,strat) ∈ sub_results do
 4     if ! real then
 5         implementation ← extendCounterStrategy(strat, s)
 6         return (⊥, implementation)
 7 impls ← map second sub_results
 8 return (⊤, compose impls)
```

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. To obtain composability of the subresults, the implementations need to agree on shared variables. We ensure this by assigning disjoint sets of output variables to the synthesis subtasks: Since every subresult only defines the behavior of the assigned output variables, the implementations are non-contradictory. Since the language alphabets of the subspecifications differ, we define the non-contradictory composition of languages:

Definition 1 (Non-Contradictory Language Composition). *Let L_1, L_2 be languages over 2^{Σ_1} and 2^{Σ_2}, respectively. The* composition of L_1 and L_2 *is defined by* $L_1 \,||\, L_2 = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in L_1 \wedge \sigma_2 \in L_2 \wedge \sigma_1 \cap \Sigma_2 = \sigma_2 \cap \Sigma_1\}$.

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

Definition 2 (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 called* independent sublanguages *of L if $L_1 \,||\, L_2 = L$ holds.*

From these two requirements, i.e., non-contradictory and independent sublanguages, equirealizability of the initial specification and the subspecifications follows. For the full proof, we refer the reader to full version [12].

Theorem 1. *Let s, s_1, s_2 be specifications with $\mathcal{L}(s) \subseteq (2^V)^\omega$, $\mathcal{L}(s_1) \subseteq (2^{V_1})^\omega$, $\mathcal{L}(s_2) \subseteq (2^{V_2})^\omega$. Recall that $I \subseteq V$ is the set of input variables. If $V_1 \cap V_2 \subseteq I$ and $V_1 \cup V_2 = V$ hold, and $\mathcal{L}(s_1)$ and $\mathcal{L}(s_2)$ are independent sublanguages of $\mathcal{L}(s)$, then s is realizable if, and only if, both s_1 and s_2 are realizable.*

Proof (Sketch). First, let s_1, s_2 be realizable and let f_1, f_2 be implementations realizing them. Let f be an implementation that acts as f_1 on $O \cap V_1$ and as f_2 on $O \cap V_2$. Since $V_1 \cap V_2 \subseteq I$ and $V_1 \cup V_2 = V$ hold, f is well-defined and defines the behavior of all outputs variables. By construction, f realizes s_1 and s_2

since f_1 and f_2 do, respectively. Since $\mathcal{L}(s_1)$ and $\mathcal{L}(s_2)$ are non-contradictory, independent sublanguages of $\mathcal{L}(s)$ by assumption, f thus realizes s.

Second, assume that s_i is unrealizable for some $i \in \{1, 2\}$. Then, there is a counterstrategy f_i^c for s_i. With the construction given in the full version [12], we can construct a counterstrategy for s from f_i^c. Hence, s is unrealizable. □

The soundness and completeness of Algorithm 1 for adequate decomposition algorithms now follows directly with Theorem 1 and the properties of such algorithms described above: They produce subspecifications that do not share output variables and that form independent sublanguages.

Theorem 2 (Soundness and Completeness). *Let s be a specification. Let $\mathcal{S} = \{s_1, \dots, s_k\}$ be a set of subspecifications with $\mathcal{L}(s_i) \subseteq (2^{V_i})^\omega$ such that $\bigcup_{1 \leq i \leq k} V_i = V$, $V_i \cap V_j \subseteq I$ for $1 \leq i, j \leq k$ with $i \neq j$, and $\mathcal{L}(s_1), \dots, \mathcal{L}(s_k)$ are independent sublanguages of $\mathcal{L}(s)$. 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. By applying Theorem 1 recursively, s_i is realizable for all $s_i \in \mathcal{S}$. Since $V_i \cap V_j \subseteq I$ for any $s_i, s_j \in \mathcal{S}$ with $i \neq j$, the implementations realizing the subspecifications are non-contradictory. Hence, Algorithm 1 returns their composition: Implementation f. Since $V_1 \cup \dots \cup V_k = V$, f defines the behavior of all outputs. By construction, f realizes all $s_i \in \mathcal{S}$. Thus, since the $\mathcal{L}(s_i)$ are non-contradictory, independent sublanguages of $\mathcal{L}(s)$, f realizes s.

Next, let s be unrealizable. Then, there exists an unrealizable subspecification $s_i \in \mathcal{S}$ and Algorithm 1 returns its extension to a counterstrategy for the whole system. The correctness of this construction is proven in the full version [12]. □

4 Decomposition of Nondeterministic Büchi Automata

To ensure soundness and completeness of modular synthesis, a decomposition algorithm has to meet the language-based adequacy conditions of Theorem 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 parallel composition instead of the parallel composition of their languages: Let $\mathcal{A}_1 = (Q_1, Q_0^1, \delta_1, F_1)$ and $\mathcal{A}_2 = (Q_2, Q_0^2, \delta_2, F_2)$ be NBAs over 2^{V_1}, 2^{V_2}, respectively. The *parallel composition of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ is defined by the NBA $\mathcal{A}_1 \,||\, \mathcal{A}_2 = (Q, Q_0, \delta, F)$ over $2^{V_1 \cup V_2}$ with $Q = Q_1 \times Q_2$, $Q_0 = Q_0^1 \times Q_0^2$, $((q_1, q_2), \boldsymbol{i}, (q_1', q_2')) \in \delta$ if, and only if, $(q_1, \boldsymbol{i} \cap V_1, q_1') \in \delta_1$ and $(q_2, \boldsymbol{i} \cap V_2, q_2') \in \delta_2$, and $F = F_1 \times F_2$. The parallel composition of NBAs reflects the parallel composition of their languages:

Lemma 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two NBAs over alphabets 2^{V_1} and 2^{V_2}, respectively. Then, $\mathcal{L}(\mathcal{A}_1 \,||\, \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \,||\, \mathcal{L}(\mathcal{A}_2)$ holds.*

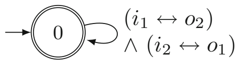
```
Algorithm 2: Automaton Decomposition
   Input: $\mathcal{A}$: NBA, inp: List Variable, out: List Variable
   Output: subautomata: List (NBA, List Variable, List Variable)
 1 if isNull checkedSubsets then
 2     checkedSubsets ← $\emptyset$
 3 subautomata ← [($\mathcal{A}$, inp, out)]
 4 foreach X ⊂ out do
 5     Y ← out\X
 6     if X ∉ checkedSubsets ∧ Y ∉ checkedSubsets then
 7         $\mathcal{A}_{\mathtt{X}} \leftarrow \mathcal{A}_{\pi(\mathtt{X} \cup \mathtt{inp})}$
 8         $\mathcal{A}_{\mathtt{Y}} \leftarrow \mathcal{A}_{\pi(\mathtt{Y} \cup \mathtt{inp})}$
 9         if $\mathcal{L}(\mathcal{A}_{\mathtt{X}} \,||\, \mathcal{A}_{\mathtt{Y}}) \subseteq \mathcal{L}(\mathcal{A})$ then
10             subautomata ← decompose($\mathcal{A}_{\mathtt{X}}$, inp, X) ++ decompose($\mathcal{A}_{\mathtt{Y}}$, inp, Y)
11             break
12     checkedSubsets ← checkedSubsets ∪ {X, Y}
13 return subautomata
```

Proof. First, let $\sigma \in \mathcal{L}(\mathcal{A}_1 \,||\, \mathcal{A}_2)$. Then, σ is an accepting run on $\mathcal{A}_1 \,||\, \mathcal{A}_2$. Hence, by definition of automaton composition, for $i \in \{1,2\}$, $\sigma \cap V_i$ is an accepting run on $\mathcal{A}_i$. Thus, $\sigma \cap V_i \in \mathcal{L}(\mathcal{A}_i)$. Since $(\sigma \cap V_1) \cap V_2 = (\sigma \cap V_2) \cap V_1$, we have $(\sigma \cap V_1) \cup (\sigma \cap V_2) \in \mathcal{L}(\mathcal{A}_1) \,||\, \mathcal{L}(\mathcal{A}_2)$. By definition of automaton composition, $\sigma \in (2^{V_1 \cup V_2})^\omega$ and thus $\sigma = (\sigma \cap V_1) \cup (\sigma \cap V_2)$. Hence, $\sigma \in \mathcal{L}(\mathcal{A}_1) \,||\, \mathcal{L}(\mathcal{A}_2)$.

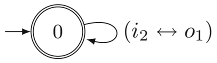
Second, let $\sigma \in \mathcal{L}(\mathcal{A}_1) \,||\, \mathcal{L}(\mathcal{A}_2)$. Then, there are $\sigma_1 \in (2^{V_1})^\omega$, $\sigma_2 \in (2^{V_2})^\omega$ with $\sigma = \sigma_1 \cup \sigma_2$ such that $\sigma_i \in \mathcal{L}(\mathcal{A}_i)$ for $i \in \{1,2\}$ and $\sigma_1 \cap V_2 = \sigma_2 \cap V_1$. Hence, σ_i is an accepting run on $\mathcal{A}_i$. Thus, by definition of automaton composition and since σ_1 and σ_2 agree on shared variables, $\sigma_1 \cup \sigma_2$ is an accepting run on $\mathcal{A}_1 \,||\, \mathcal{A}_2$. Thus, $\sigma_1 \cup \sigma_2 \in \mathcal{L}(\mathcal{A}_1 \,||\, \mathcal{A}_2)$ and hence $\sigma \in \mathcal{L}(\mathcal{A}_1 \,||\, \mathcal{A}_2)$ holds. □

Using the above lemma, we can formalize the independent sublanguage criterion on NBAs directly: Two automata $\mathcal{A}_1$, $\mathcal{A}_2$ are *independent subautomata of* $\mathcal{A}$ *if* $\mathcal{A} = \mathcal{A}_1 \,||\, \mathcal{A}_2$. To apply Theorem 1, the alphabets of the subautomata may not share output variables. Our decomposition algorithm achieves this by constructing the subautomata from the initial automaton by projecting to disjoint sets of outputs. Intuitively, the projection to a set X abstracts from the variables outside of X. Hence, it only captures the parts of the initial specification concerning the variables in X. Formally: Let $\mathcal{A} = (Q, Q_0, \delta, F)$ be an NBA over alphabet 2^V and let $X \subset V$. The *projection of* $\mathcal{A}$ *to* X is the NBA $\mathcal{A}_{\pi(X)} = (Q, Q_0, \pi_X(\delta), F)$ over 2^X with $\pi_X(\delta) = \{(q, a, q') \in Q \times 2^X \times Q \mid \exists\, b \in 2^{V \setminus X}.\, (q, a \cup b, q') \in \delta\}$.

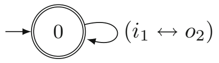
The decomposition algorithm for NBAs is described in Algorithm 2. It is a recursive algorithm that, starting with the initial automaton $\mathcal{A}$, guesses a subset X of the output variables out. It abstracts from the output variables outside of X by building the projection $\mathcal{A}_{\mathtt{X}}$ of $\mathcal{A}$ to X ∪ inp, where inp is the set of input variables. Similarly, it builds the projection $\mathcal{A}_{\mathtt{Y}}$ of $\mathcal{A}$ to Y := (out \ X) ∪ inp. By construction of $\mathcal{A}_{\mathtt{X}}$ and $\mathcal{A}_{\mathtt{Y}}$ and since both X ∩ Y = ∅ and X ∪ Y = out hold, we


0
$(i_1 \leftrightarrow o_2)$
$\wedge\ (i_2 \leftrightarrow o_1)$

(a) NBA $\mathcal{A}$



(b) NBA $\mathcal{A}_{\pi(V_1)}$



(c) NBA $\mathcal{A}_{\pi(V_2)}$

Fig. 1. NBA $\mathcal{A}$ for the *shift_2* specification and its projections $\mathcal{A}_{\pi(V_1)}$ and $\mathcal{A}_{\pi(V_2)}$ to $V_1 = \{i_1, i_2, o_1\}$ and $V_2 = \{i_1, i_2, o_2\}$. All states are accepting.

have $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_\mathtt{X} \mid\mid \mathcal{A}_\mathtt{Y})$. Hence, if $\mathcal{L}(\mathcal{A}_\mathtt{X} \mid\mid \mathcal{A}_\mathtt{Y}) \subseteq \mathcal{L}(\mathcal{A})$ holds, then $\mathcal{A}_\mathtt{X} \mid\mid \mathcal{A}_\mathtt{Y}$ is equivalent to $\mathcal{A}$ and therefore $\mathcal{L}(\mathcal{A}_\mathtt{X})$ and $\mathcal{L}(\mathcal{A}_\mathtt{Y})$ are independent sublanguages of $\mathcal{L}(\mathcal{A})$. Thus, since $\mathtt{X} \cap \mathtt{Y} = \emptyset$ holds, $\mathcal{A}_\mathtt{X}$ and $\mathcal{A}_\mathtt{Y}$ are a valid decomposition of $\mathcal{A}$. The subautomata are then decomposed recursively. If no further decomposition is possible, the algorithm returns the subautomata. By only considering unexplored subsets of output variables, no subset combination $\mathtt{X}, \mathtt{Y}$ is checked twice.

As an example for the decomposition algorithm, consider the specification $\varphi = \square((i_1 \leftrightarrow o_2) \wedge (i_2 \leftrightarrow o_1))$ for inputs $I = \{i_1, i_2\}$ and outputs $O = \{o_1, o_2\}$. The NBA $\mathcal{A}$ that accepts $\mathcal{L}(\varphi)$ is depicted in Fig. 1a. The subautomata obtained with Algorithm 2 are shown in Figs. 1b and 1c. Clearly, $V_1 \cap V_2 \subseteq I$ holds. Moreover, their parallel composition accepts exactly those words that satisfy φ. For a slightly modified specification $\varphi' = \square((i_1 \leftrightarrow o_2) \vee (i_2 \leftrightarrow o_1))$, however, Algorithm 2 does not decompose the NBA $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\varphi')$: In fact, the only possible decomposition is $\mathtt{X} = \{o_1\}$, $\mathtt{Y} = \{o_2\}$ (or vice-versa), yielding NBAs $\mathcal{A}'_\mathtt{X}$ and $\mathcal{A}'_\mathtt{Y}$ that accept every infinite word. Clearly, $\mathcal{L}(\mathcal{A}'_\mathtt{X} \mid\mid \mathcal{A}'_\mathtt{Y}) \not\subseteq \mathcal{L}(\mathcal{A}')$ since $\mathcal{L}(\mathcal{A}'_\mathtt{X} \mid\mid \mathcal{A}'_\mathtt{Y}) = (2^{I \cup O})^\omega$ and hence $\mathcal{A}'_\mathtt{X}$ and $\mathcal{A}'_\mathtt{Y}$ 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 directly from the construction of the subautomata, Lemma 1, and Theorem 1. The proof is given in the full version [12].

Theorem 3. *Let $\mathcal{A}$ be an NBA over alphabet 2^V. Algorithm 2 terminates on $\mathcal{A}$ with a set $\mathcal{S} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ of NBAs with $\mathcal{L}(\mathcal{A}_i) \subseteq (2^{V_i})^\omega$, 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 $\mathcal{A}$ is realizable if, and only if, for all $\mathcal{A}_i \in \mathcal{S}$, $\mathcal{A}_i$ is realizable.*

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

Theorem 4. *Let $\mathcal{A}$ be an NBA and let $\mathcal{S}$ be the set of NBAs that Algorithm 2 returns on input $\mathcal{A}$. Then, for each $\mathcal{A}_i \in \mathcal{S}$ over alphabet 2^{V_i}, there are no NBAs $\mathcal{A}'$, $\mathcal{A}''$ over alphabets $2^{V'}$ and $2^{V''}$ with $V_i = V' \cup V''$ such that $\mathcal{A}_i = \mathcal{A}' \mid\mid \mathcal{A}''$.*

Hence, Algorithm 2 yields *perfect* decompositions and is semantically precise. Yet, it performs several expensive automaton operations such as projection, composition, and language containment checks. For large automata, this is infeasible. For specifications given as LTL formulas, we thus present an approximate decomposition algorithm in the next section that does not yield non-decomposable subspecifications, but that 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 $V = \mathit{prop}(\varphi)$ holds for the initial specification φ. Note that any implementation for the variables in $\mathit{prop}(\varphi)$ can easily be extended to one for the variables in V if $\mathit{prop}(\varphi) \subset V$ by ignoring the inputs in $I \setminus \mathit{prop}(\varphi)$ and by choosing arbitrary valuations for the outputs in $O \setminus \mathit{prop}(\varphi)$.

The main idea of the decomposition algorithm is to rewrite the initial LTL formula φ into a conjunctive form $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_k$ with as many top-level conjuncts as possible by applying distributivity and pushing temporal operators inwards whenever possible. Then, we build subspecifications 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 independent conjuncts, the languages of the conjuncts are independent sublanguages of the language of the whole formula:

Lemma 2. *Let $\varphi = \varphi_1 \wedge \varphi_2$ be an LTL formula over Σ. Let $\mathcal{L}(\varphi_1) \in (2^{\Sigma_1})^\omega$, $\mathcal{L}(\varphi_2) \in (2^{\Sigma_2})^\omega$ be the languages of φ_1 and φ_2 over Σ_1 and Σ_2, respectively, with $\Sigma_1 \cup \Sigma_2 = V$. 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, $\sigma \in \mathcal{L}(\varphi_i)$ for all $i \in \{1, 2\}$. Since $\mathit{prop}(\varphi_i) \subseteq \Sigma_i$ holds by definition and since the satisfaction of an LTL formula does only depend on the valuations of the variables in $\mathit{prop}(\varphi_i)$, we have $\sigma \cap \Sigma_i \in \mathcal{L}(\varphi_i)$. Since clearly $(\sigma \cap \Sigma_1) \cap \Sigma_2 = (\sigma \cap \Sigma_2) \cap \Sigma_1$ holds, $(\sigma \cap \Sigma_1) \cup (\sigma \cap \Sigma_2) \in \mathcal{L}(\varphi_1) \,||\, \mathcal{L}(\varphi_2)$. Since $\Sigma_1 \cup \Sigma_2 = \Sigma$, $\sigma = (\sigma \cap \Sigma_1) \cup (\sigma \cap \Sigma_2)$ and hence $\sigma \in \mathcal{L}(\varphi_1) \,||\, \mathcal{L}(\varphi_2)$.

Next, let $\sigma \in \mathcal{L}(\varphi_1) \,||\, \mathcal{L}(\varphi_2)$. Then, there are words $\sigma_1 \in \mathcal{L}(\varphi_1)$, $\sigma_2 \in \mathcal{L}(\varphi_2)$ with $\sigma_1 \cap \Sigma_2 = \sigma_2 \cap \Sigma_1$ and $\sigma = \sigma_1 \cup \sigma_2$. Since σ_1 and σ_2 agree on shared variables, $\sigma \in \mathcal{L}(\varphi_1)$ and $\sigma \in \mathcal{L}(\varphi_2)$ follows. Hence, $\sigma \in \mathcal{L}(\varphi_1 \wedge \varphi_2)$. $\square$

Our decomposition algorithm then ensures that different subspecifications share only input variables by merging conjuncts that share output variables into the same subspecification. Then, equirealizability of the initial formula and the subformulas follows directly from Theorem 1 and Lemma 2:

Corollary 1. *Let $\varphi = \varphi_1 \wedge \varphi_2$ be an LTL formula over V with conjuncts φ_1, φ_2 over V_1, V_2, respectively, with $V_1 \cup V_2 = V$ and $V_1 \cap V_2 \subseteq I$. Then, φ is realizable if, and only if, both φ_1 and φ_2 are realizable.*

To determine conjuncts of an LTL formula $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ that share variables, we build the *dependency graph* $\mathcal{D}_\varphi = (V, E)$ of the output variables, where $V = O$ and $(a, b) \in E$ if, and only if, $a \in \mathit{prop}(\varphi_i)$ and $b \in \mathit{prop}(\varphi_i)$ for some $1 \leq i \leq n$. Intuitively, 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 either occur in the same conjunct or that they occur in conjuncts that are connected by other output

Algorithm 3: LTL Decomposition

```
    Input: $\varphi$: LTL, inp: List Variable, out: List Variable
    Output: specs: List (LTL, List Variable, List Variable)
1   $\varphi \leftarrow$ rewrite($\varphi$)
2   formulas $\leftarrow$ removeTopLevelConjunction($\varphi$)
3   graph $\leftarrow$ buildDependencyGraph($\varphi$, out)
4   components $\leftarrow$ graph.connectedComponents()
5   specs $\leftarrow$ new LTL[|components|+1] // initialized with true
6   foreach $\psi \in$ formulas do
7       propositions $\leftarrow$ getPropositions($\psi$)
8       foreach (spec,set) $\in$ zip specs (components ++ [inp]) do
9           if propositions $\cap$ set $\neq \emptyset$ then
10              spec.And($\psi$)
11              break
12  return map ($\lambda\varphi \rightarrow$ ($\varphi$, inputs($\varphi$), outputs($\varphi$))) specs
```

variables. Hence, to ensure that subspecifications do not share output variables, conjuncts containing a or b need to be assigned to the same subspecification. Output variables that are contained in different connected 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 decomposed into subspecifications. First, the formula is rewritten into conjunctive form. Then, the dependency graph is built and the connected components are computed. 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. Considering the input variables is necessary to assign every conjunct, including input-only ones, 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, the algorithm assigns the inputs and outputs occurring in φ_i to the subspecification φ_i. While restricting the inputs is not necessary for correctness, it may improve the runtime of the corresponding synthesis task.

Soundness and completeness of modular synthesis with Algorithm 3 as a decomposition algorithm for LTL formulas follows directly from Corollary 1 if the subspecifications do not share any output variables:

Theorem 5. *Let φ be an LTL formula over V. Then, Algorithm 3 terminates on φ with a set $\mathcal{S} = \{\varphi_1, \dots, \varphi_k\}$ of LTL formulas with $\mathcal{L}(\varphi_i) \in (2^{V_i})^\omega$ such that $V_i \cap V_j \subseteq I$ for $1 \leq i, j \leq k$ 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 $\varphi_i \in \mathcal{S}$, φ_i is realizable.*

Proof. Since an output variable is part of exactly one connected component and since all conjuncts containing an output are contained in the same subspecification, every output is part of exactly one subspecification. Therefore, $V_i \cap V_j \subseteq I$ holds for $1 \leq i, j \leq k$ with $i \neq j$. Moreover, the last component added in Line 8 contains all inputs. Hence, all variables that occur in a conjunct of φ are featured in at least one subspecification. Thus, $\bigcup_{1 \leq i \leq k} V_i = \mathit{prop}(\varphi)$ holds and hence, since $V = \mathit{prop}(\varphi)$ by assumption, $\bigcup_{1 \leq i \leq k} V_i = V$ follows. Therefore, equirealizability of φ and the formulas in $\mathcal{S}$ directly follows with Corollary 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 assumption-guarantee format, i.e., $\varphi = \bigwedge_{i=1}^{m} \varphi_i \rightarrow \bigwedge_{j=1}^{n} \psi_j$, to a conjunctive form, the conjuncts contain both assumptions φ_i and guarantees ψ_j. Hence, if $a, b \in O$ occur in assumption φ_i and guarantee ψ_j, respectively, they are dependent. Thus, all conjuncts featuring a or b are contained in the same subspecification according to Algorithm 3. Yet, ψ_j might be realizable even without φ_i. An algorithm accounting for this might yield further decompositions.

In the following, we present a criterion for dropping assumptions in a sound and complete fashion. 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 $\neg\varphi$ is realizable, equirealizability is not guaranteed when dropping the assumption. For instance, consider the formula $\varphi = \diamondsuit(i_1 \wedge o_1) \rightarrow \square(i_2 \wedge o_2)$, where $I = \{i_1, i_2\}$ and $O = \{o_1, o_2\}$. Although assumption and guarantee do not share any variables, the assumption cannot be dropped: An implementation that never sets o_1 to *true* satisfies φ but $\square(i_2 \wedge o_2)$ 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 $((\square i_1 \rightarrow i_2) \wedge (\neg\square i_1 \rightarrow i_3) \wedge (i_2 \leftrightarrow i_4) \wedge (i_3 \leftrightarrow \neg i_4)) \rightarrow (\square i_1 \leftrightarrow o)$, where $I = \{i_1, i_2, i_3, i_4\}$ and $O = \{o\}$, no assumption can be dropped: Otherwise the information about the global behavior of i_1, which is crucial for the existence of an implementation, is incomplete. This leads to the following criterion for dropping assumptions. For the full proof, we refer to the full version [12].

Lemma 3 (Dropping Assumptions). *Let $\varphi = (\varphi_1 \wedge \varphi_2) \rightarrow \psi$ be an LTL formula with $\mathit{prop}(\varphi_1) \cap \mathit{prop}(\varphi_2) = \emptyset$ and $\mathit{prop}(\varphi_2) \cap \mathit{prop}(\psi) = \emptyset$. Let $\neg\varphi_2$ be unrealizable. Then, $\varphi_1 \rightarrow \psi$ is realizable if, and only if, φ is realizable.*

Proof (Sketch). First, assume that $\varphi' := \varphi_1 \rightarrow \psi$ is realizable and let f be an implementation realizing it. Clearly, a strategy that ignores inputs outside of $\mathit{prop}(\varphi')$, behaves as f on outputs in $\mathit{prop}(\varphi')$, and chooses arbitrary valuations for the outputs outside of $\mathit{prop}(\varphi')$, realizes $(\varphi_1 \wedge \varphi_2) \rightarrow \psi$.

Next, assume that $(\varphi_1 \wedge \varphi_2) \rightarrow \psi$ is realizable and let f be an implementation realizing it. Since $\neg\varphi_2$ is unrealizable by assumption, there exists a counterstrategy f_2^c with $\mathcal{C}(f_2^c) \subseteq \mathcal{L}(\varphi_2)$. For every $\sigma \in (2^{\mathit{prop}(\varphi')})^\omega$, we can construct a word $\hat{\sigma} \in (2^V)^\omega$ with f_2^c that is equivalent to σ on the variables in $\mathit{prop}(\varphi')$ but satisfies φ_2. Let g be an implementation that for every input σ behaves as f on $\hat{\sigma}$. Since g behaves as f but ensures that φ_2 is satisfied, it realizes φ'. □

By dropping assumptions, we are able to decompose LTL formulas of the form $\varphi = \bigwedge_{i=1}^{m} \varphi_i \rightarrow \bigwedge_{j=1}^{n} \psi_j$ in further cases: Intuitively, we rewrite φ to $\bigwedge_{j=1}^{n}(\bigwedge_{i=1}^{m} \varphi_i \rightarrow \psi_j)$ and then drop assumptions for the individual guarantees. If the resulting subspecifications only share input variables, they are equirealizable to φ. For the full proof, we refer to the full version of this paper [12].

Theorem 6. *Let $\varphi = (\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow (\psi_1 \wedge \psi_2)$ be an LTL formula over V, where $prop(\varphi_3) \subseteq I$ and $prop(\psi_1) \cap prop(\psi_2) \subseteq I$. Let $prop(\varphi_1) \cap prop(\varphi_2) = \emptyset$, $prop(\varphi_1) \cap prop(\varphi_3) = \emptyset$, $prop(\varphi_2) \cap prop(\varphi_3) = \emptyset$, and $prop(\varphi_i) \cap prop(\psi_{3-i}) = \emptyset$ for $i \in \{1, 2\}$. Let $\neg(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$ be unrealizable. Then, φ is realizable if, and only if, both $\varphi' = (\varphi_1 \wedge \varphi_3) \rightarrow \psi_1$ and $\varphi'' = (\varphi_2 \wedge \varphi_3) \rightarrow \psi_2$ are realizable.*

Proof (Sketch). First, let φ be realizable and let f be an implementation realizing it. Clearly, f realizes $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_i$ for all $i \in \{1, 2\}$ as well. By Lemma 3, $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_i$ and $(\varphi_i \wedge \varphi_3) \rightarrow \psi_i$ are equirealizable since φ_1, φ_2, and φ_3 do not share any variables and φ_{3-i} and ψ_i only share input variables by assumption. Thus, there are implementations realizing φ' and φ''.

Next, let φ' and φ'' be realizable and let f_1, f_2 be implementations realizing them. Let f be an implementation that acts as f_1 on the variables in $prop(\varphi')$ and as f_2 on the variables in $prop(\varphi'')$.The formulas only share variables in $prop(\varphi_3)$ and thus only input variables. Hence, f is well-defined. By construction, f realizes both φ' and φ''. Thus, since $\varphi' \wedge \varphi''$ implies φ, f realizes φ. $\square$

Analyzing assumptions thus allows for decomposing LTL formulas in further cases and still ensures soundness and completeness of modular synthesis. A modified LTL decomposition algorithm needs to identify variables that cannot be shared safely among subspecifications. If an assumption contains such variables, it is *bound* to guarantees. Otherwise, it is *free*. Guarantees are decomposed as in Algorithm 3. Then, bounded assumptions are added to the subspecifications of their respective guarantees. Free assumptions can be added to all subspecifications. To obtain small subspecifications, though, further optimizations can be used. A detailed description of the algorithm is given in the full version [12].

Note that the decomposition algorithm does not check for possible violations of assumptions. Instead, we slightly modify the modular synthesis algorithm: Before decomposing, we perform synthesis on the negated assumptions. If it returns realizable, it is possible to violate an assumption. The implementation is extended to an implementation for the whole specification that violates the assumptions and thus realizes the specification. Otherwise, if the negated assumptions are unrealizable, the conditions of Theorem 6 are satisfied. Hence, we can use the decomposition algorithm and proceed as in Algorithm 1.

6 Experimental Evaluation

We implemented the modular synthesis algorithm as well as the decomposition approaches and evaluated them on the 346 publicly available SYNTCOMP [16] benchmarks. Note that only 207 of the benchmarks have more than one output

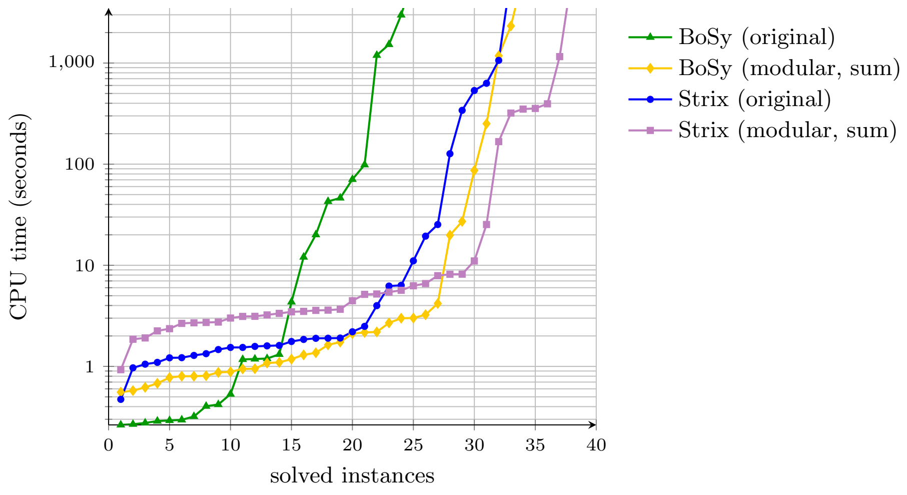


Fig. 2. Comparison of the performance of modular and non-compositional synthesis with BoSy and Strix on the decomposable SYNTCOMP benchmarks. For the modular approach, the accumulated time for all synthesis tasks is depicted.

variable and are therefore realistic candidates for decomposition. The automaton decomposition algorithm utilizes the Spot (2.9.6) automaton library [7] and the LTL decomposition relies on SyFCo (1.2.1.1) [17] for formula transformations. We first decompose the specification and then run synthesis on the resulting subspecifications. We compare the CPU Time, Gates, and Latches for the original specification to the sum of the corresponding attributes of all subspecifications. Thus, we calculate the runtime for sequential modular synthesis. Parallelization of the synthesis tasks may further reduce the runtime.

6.1 LTL Decomposition

LTL decomposition with optimized assumption handling terminates on all benchmarks in less than 26 ms. Thus, even for non-decomposable specifications, the overhead is negligible. The algorithm 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.

We evaluate our modular synthesis approach with two state-of-the-art synthesis tools: BoSy [9], a bounded synthesis tool, and Strix [22], a game-based synthesis tool, both in their 2019 release. We used a machine with a 3.6 GHz quad-core Intel Xeon processor and 32 GB RAM and a timeout of 60 min. In Fig. 2, the comparison of the accumulated runtimes of the synthesis of the subspecifications and the original formula is shown for the decomposable benchmarks. For both BoSy and Strix, decomposition generates a slight overhead for small specifications. For larger and more complex benchmarks, however, modular synthesis decreases the execution time significantly, often by an order of magnitude or more. Note that due to

Table 1. Synthesis time (in seconds) of BoSy and Strix for non-compositional and modular synthesis on exemplary SYNTCOMP benchmarks.

Benchmark		Original		Modular	
	# subspec.	BoSy	Strix	BoSy	Strix
Cockpitboard	8	1526.32	11.06	**2.108**	8.168
Gamelogic	4	TO	1062.27	TO	**25.292**
LedMatrix	3	TO	TO	TO	**1156.68**
Radarboard	11	TO	126.808	**3.008**	11.04
Zoo10	2	1.316	1.54	**0.884**	2.744
generalized_buffer_2	2	70.71	534.732	**4.188**	7.892
generalized_buffer_3	2	TO	TO	**27.136**	319.988
shift_8	8	**0.404**	1.336	2.168	3.6
shift_10	10	**1.172**	1.896	2.692	4.464
shift_12	12	4.336	6.232	**3.244**	5.428

the negligible runtime of specification decomposition, the plot looks similar when considering all SYNTCOMP benchmarks.

Table 1 shows the running times of BoSy and Strix for modular and non-compositional synthesis on exemplary benchmarks. 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 2020, 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 Table 2, the number of gates and latches 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. Gates and latches of the solutions of BoSy and Strix for non-compositional and modular synthesis on exemplary SYNTCOMP benchmarks.

Benchmark	Gates				Latches			
	Original		Modular		Original		Modular	
	BoSy	Strix	BoSy	Strix	BoSy	Strix	BoSy	Strix
Cockpitboard	11	**7**	25	10	1	**0**	8	**0**
Gamelogic	–	26	–	**21**	–	**2**	–	**2**
LedMatrix	–	–	–	**97**	–	–	–	**5**
Radarboard	–	**6**	19	**6**	–	**0**	11	**0**
Zoo10	14	15	15	**13**	**1**	2	2	2
generalized_buffer_2	**3**	12	**3**	11	69	47134	**14**	557
generalized_buffer_3	–	–	**20**	3772	–	–	**3**	14
shift_8	8	**0**	8	7	1	**0**	8	**0**
shift_10	10	**0**	10	9	1	**0**	10	**0**
shift_12	12	**0**	12	11	1	**0**	12	**0**

6.2 Automata Decomposition

Besides LTL specifications, Strix also accepts specifications given as deterministic parity automata (DPAs) in extended HOA format [23], an automaton format well-suited for synthesis. Thus, our implementation performs Algorithm 2, converts the resulting automata to DPAs and synthesizes solutions with Strix.

For 235 out of the 346 SYNTCOMP benchmarks, decomposition terminated within 10 min yielding several subspecifications or proving that the specification is not decomposable. In 79 of the other cases, the tool timed out and in the remaining 32 cases it reached the memory limit of 16 GB or the internal limits of Spot. Note, however, that for 81 specifications even plain DPA generation fails. Thus, while the automaton decomposition algorithm yields more fine-grained decompositions than the approximate LTL approach, it becomes infeasible when the specifications grow. Hence, the advantage of smaller synthesis subtasks cannot pay off. 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.

For 43 specifications, the automaton approach yields decompositions and many of them consist of four or more subspecifications. For 22 of these specifications, the LTL approach yields a decomposition as well. Yet, they differ in most cases, as the automaton approach yields more fine-grained decompositions.

Recall that only 207 SYNTCOMP benchmarks are realistic candidates for decomposition. The automaton approach proves that 90 of those specifications (43.6%) are not decomposable. Thus, our implementations yield decompositions for 33.33% (LTL) and 36.75% (Automaton) of the potentially decomposable specifications. We observed that decomposition works exceptionally well for specifications that stem from real system designs, for instance the Syntroids [14] case study, indicating that modular synthesis is particularly beneficial in practice.

7 Conclusions

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 nondeterministic Büchi automata, and an approximate algorithm for LTL formulas. 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. Moreover, they improve their synthesis times on complex specifications notably. This shows that decomposing the specification is a game-changer for practical 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. Moreover, parallelizing the individual synthesis tasks may expand 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.

References

1. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
2. Bloem, R., Chatterjee, K., Jacobs, S., Könighofer, R.: Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 517–532. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_50
3. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_21
4. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science. LICS 1989, pp. 353–362. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39190

5. Damm, W., Finkbeiner, B.: Does it pay to extend the perimeter of a world model? In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 12–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_4
6. Dureja, R., Rozier, K.Y.: More scalable LTL model checking via discovering design-space dependencies (D^3). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 309–327. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_17
7. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
8. Ehlers, R.: Unbeast: symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_25
9. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 325–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_17
10. Filiot, E., Jin, N., Raskin, J.-F.: Compositional algorithms for LTL synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_10
11. Finkbeiner, B.: 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, pp. 72–98. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-72
12. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis (full version). CoRR abs/2103.08459 (2021). https://arxiv.org/abs/2103.08459
13. Finkbeiner, B., Passing, N.: Dependency-based compositional synthesis. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 447–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_25
14. Geier, G., Heim, P., Klein, F., Finkbeiner, B.: Syntroids: synthesizing a game for FPGAs using temporal logic specifications. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, Proceedings, pp. 138–146. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894261
15. Jacobs, S., Bloem, R.: The reactive synthesis competition: SYNTCOMP 2016 and beyond. In: Piskac, R., Dimitrova, R. (eds.) Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings. EPTCS, vol. 229, pp. 133–148 (2016). https://doi.org/10.4204/EPTCS.229.11
16. Jacobs, S., et al.: The 5th reactive synthesis competition (SYNTCOMP 2018): benchmarks, participants & results. CoRR abs/1904.07736 (2019). http://arxiv.org/abs/1904.07736
17. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Piskac, R., Dimitrova, R. (eds.) Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings. EPTCS, vol. 229, pp. 112–132 (2016). https://doi.org/10.4204/EPTCS.229.10
18. Jobstmann, B.: Applications and optimizations for LTL synthesis. Ph.D. thesis, Graz University of Technology (2007)
19. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_6

20. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS) 2005, Proceedings, pp. 531–542. IEEE Computer Society (2005)
21. Majumdar, R., Mallik, K., Schmuck, A., Zufferey, D.: Assume-guarantee distributed synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(11), 3215–3226 (2020). https://doi.org/10.1109/TCAD.2020.3012641
22. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 578–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_31
23. Pérez, G.A.: The extended HOA format for synthesis. CoRR abs/1912.05793 (2019). http://arxiv.org/abs/1912.05793
24. Pnueli, A.: The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages 1989, pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
26. de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.): COMPOS 1997. LNCS, vol. 1536. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49213-5
27. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. Int. J. Softw. Tools Technol. Transf. **15**(5–6), 433–454 (2013). https://doi.org/10.1007/s10009-012-0224-3