Dependency-based Compositional Synthesis[⋆]

Bernd Finkbeiner[0000−0002−4280−8441] and Noemi Passing[0000−0001−7781−043X]

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner, noemi.passing}@cispa.saarland

Abstract. Despite many recent advances, reactive synthesis is still not really a practical technique. The grand challenge is to scale from small transition systems, where synthesis performs well, to complex multi-component designs. Compositional methods, such as the construction of dominant strategies for individual components, reduce the complexity significantly, but are usually not applicable without extensively rewriting the specification. In this paper, we present a refinement of compositional synthesis that does not require such an intervention. Our algorithm decomposes the system into a sequence of components, such that every component has a strategy that is dominant, i.e., performs at least as good as any possible alternative, provided that the preceding components follow their (already synthesized) strategies. The decomposition of the system is based on a dependency analysis, for which we provide semantic and syntactic techniques. We establish the soundness and completeness of the approach and report on encouraging experimental results.

1 Introduction

Compositionality breaks the analysis of a complex system into several smaller tasks over individual components. It has long been recognized as the key technique that makes a "significant difference" [16] for the scalability of verification algorithms. In synthesis, it has proven much harder to develop successful compositional techniques. In a nutshell, synthesis corresponds to finding a winning strategy for the system in a game against its environment. In compositional synthesis, the system player controls an individual component, the environment player all remaining components [9]. In practice, however, a winning strategy rarely exists for an individual component, because the specification can usually only be satisfied if several components collaborate.

Remorsefree dominance [3], a weaker notion than winning, accounts for such situations. Intuitively, a dominant strategy is allowed to violate the specification as long as no other strategy would have satisfied it in the same situation. In other words, if the violation is the fault of the environment, we do not blame the component. Looking for strategies that are dominant, rather than winning,

⋆ 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).

allows us to find strategies that do not necessarily satisfy the specification for all input sequences, but satisfy the specification for sequences that are *realistic* in the sense that they might actually occur in a system that is built from components that all do their best to satisfy the specification.

For safety specifications, it was shown that dominance is a compositional notion: the composition of two dominant strategies is again dominant. Furthermore, if a winning strategy exists, then all dominant strategies are winning. This directly leads to a compositional synthesis approach that synthesizes individual dominant strategies [4]. In general, however, there is no guarantee that a dominant strategy exists. Often, a component A depends on the well-behavior of another component B in the sense that A needs to anticipate some future action of B. In such situations, there is no dominant strategy for A alone since the decision which strategy is best for A depends on the specific strategy for B.

In this paper, we address this problem with an *incremental* synthesis approach. Like in standard compositional synthesis, we split the system into components. However, we do not try to find dominant strategies for each component individually. Rather, we proceed in an incremental fashion such that each component can already assume a particular strategy for the previously synthesized components. We call the order, in which the components are constructed, the *synthesis order*. Instead of requiring the existence of dominant strategies for all components, we only require the existence of a dominant strategy *under the assumption* of the previous strategies. Similar to standard compositional synthesis, this approach reduces the complexity of synthesis by decomposing the system; additionally, it overcomes the problem that dominant strategies generally do not exist for all components without relying on other strategies.

The key question now is how to find the synthesis order. We propose two methods that offer different trade-offs between precision and computational cost. The first method is based on a semantic dependency analysis of the output variables of the system. We build equivalence classes of variables based on cyclic dependencies, which then form the components of the system. The synthesis order is defined on the dependencies between the components, resolving dependencies that prevent the existence of dominant strategies. The second method is based on a syntactic analysis of the specification, which conservatively overapproximates the semantic dependencies.

We have implemented a prototype of the incremental synthesis algorithm and compare it to the state-of-the-art synthesis tool BoSy [6] on scalable benchmarks. The results are very encouraging: our algorithm clearly outperforms classical synthesis for larger systems.

Proofs and the benchmark specifications are provided in the full version [8].

Related Work. Kupferman et al. introduce a safraless compositional synthesis algorithm transforming the synthesis problem into an emptiness check on Büchi tree automata [13]. Kugler and Sittal introduce two compositional algorithms for synthesis from Live Sequence Charts specifications [12]. Yet, neither of them is sound and complete. While they briefly describe a sound and complete extension of their algorithms, they did not implement it. Filiot et al. introduce a compositional synthesis algorithm for LTL specifications [7] based on the composition of safety games. Moreover, they introduce a non-complete heuristic for dropping conjuncts of the specification. All of the above approaches search for winning strategies and thus fail if cooperation between the components is needed.

The notion of remorsefree dominance was first introduced in the setting of reactive synthesis by Damm and Finkbeiner [3]. They introduce a compositional synthesis algorithm for safety properties based on dominant strategies [4].

In the setting of controller synthesis, Baier et al. present an algorithm that incrementally synthesizes so-called most general controllers and builds their parallel composition in order to synthesize the next one [1]. In contrast to our approach, they do not decompose the system in separate components. Incremental synthesis is only used to handle cascades of objectives in an online fashion.

2 Motivating Example

In safety-critical systems such as self-driving cars, correctness of the implementation with respect to a given specification is crucial. Hence, they are an obvious target for synthesis. However, a self-driving car consists of several components that interact with each other, leading to enormous state spaces when synthesized together. While a compositional approach may reduce the complexity, in most scenarios there are neither winning nor dominant strategies for the separate components. Consider a specification for a gearing unit and an acceleration unit of a self-driving car. The latter one is required to decelerate before curves and to not accelerate in curves. To prevent traffic jams, the car is required to accelerate eventually if no curve is ahead. In order to safe fuel, it should not always accelerate or decelerate. This can be specified in LTL as follows:

$$\begin{aligned}\varphi_{acc} = {} & \square(\mathit{curve\_ahead} \rightarrow \bigcirc\, \mathit{dec}) \land \square(\mathit{in\_curve} \rightarrow \bigcirc \neg \mathit{acc}) \land \square \Diamond\, \mathit{keep} \\ & \land \square((\neg \mathit{in\_curve} \land \neg \mathit{curve\_ahead}) \rightarrow \Diamond\, \mathit{acc}) \land \square \neg(\mathit{acc} \land \mathit{dec}) \\ & \land \square \neg(\mathit{acc} \land \mathit{keep}) \land \square \neg(\mathit{dec} \land \mathit{keep}) \land \square(\mathit{acc} \lor \mathit{dec} \lor \mathit{keep}),\end{aligned}$$

where *curve_ahead* and *in_curve* are input variables denoting whether a curve is ahead or whether the car is in a curve, respectively. The output variables are *acc* and *dec*, denoting acceleration and deceleration, and *keep*, denoting that the current speed is kept. Note that φ_{acc} is only realizable if we assume that a curve is not followed by another one with only one step in between infinitely often.

The gearing unit can choose between two gears. It is required to use the smaller gear when the car is accelerating and the higher gear if the car reaches a steady speed after accelerating. This can be specified in LTL as follows, where gear_i are output variables denoting whether the first or the second gear is used:

$$\begin{aligned}\varphi_{gear} = {} & \square((\mathit{acc} \land \bigcirc\, \mathit{acc}) \rightarrow \bigcirc\bigcirc\, \mathit{gear}_1) \land \square((\mathit{acc} \land \bigcirc\, \mathit{keep}) \rightarrow \bigcirc\bigcirc\, \mathit{gear}_2) \\ & \land \square \neg(\mathit{gear}_1 \land \mathit{gear}_2) \land \square(\mathit{gear}_1 \lor \mathit{gear}_2).\end{aligned}$$

When synthesizing a strategy s for the acceleration unit, it does not suffice to consider only φ_{acc} since s affects the gearing unit. Yet, there is clearly no winning strategy for $\varphi_{car} := \varphi_{acc} \land \varphi_{gear}$ when considering the acceleration unit separately. There is no dominant strategy either: As long as the car accelerates after a curve, the conjunct $\Box((\neg \mathit{in\_curve} \land \neg \mathit{curve\_ahead}) \rightarrow \Diamond\, \mathit{acc})$ is satisfied. If the gearing unit does not react correctly, φ_{car} is violated. Yet, an alternative strategy for the acceleration unit that accelerates at a different point in time at which the gearing unit reacts correctly, satisfies φ_{car}. Thus, neither a compositional approach using winning strategies, nor one using dominant strategies, is able to synthesize strategies for the components of the self-driving car.

However, the lack of a dominant strategy for the acceleration unit is only due to the uncertainty whether the gearing unit will comply with the acceleration strategy. The only dominant strategy for the gearing unit is to react correctly to the change of speed. Hence, providing this knowledge to the acceleration unit by synthesizing the strategy for the gearing unit beforehand and making it available, yields a dominant and even winning strategy for the acceleration unit. Thus, synthesizing the components *incrementally* instead of *compositionally* allows for separate strategies even if there is a dependency between the components.

3 Preliminaries

LTL. Linear-time temporal logic (LTL) 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 $\varphi, \psi ::= a \mid \neg\varphi \mid \varphi \lor \psi \mid \varphi \land \psi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \psi \mid \varphi \mathcal{W} \psi$. We define the abbreviations $\mathit{true} := a \lor \neg a$, $\mathit{false} := \neg \mathit{true}$, $\Diamond \varphi = \mathit{true}\, \mathcal{U}\, \varphi$, and $\Box \varphi = \neg \Diamond \neg \varphi$ as usual and use the standard semantics. The language $\mathcal{L}(\varphi)$ of a formula φ is the set of infinite words that satisfy φ.

Automata. Given a finite alphabet Σ, a universal co-Büchi automaton is a tuple $\mathcal{A} = (Q, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times 2^{\Sigma} \times Q$ is a transition relation, and $F \subseteq Q$ is a set of rejecting states. Given an infinite word $\sigma = \sigma_0\sigma_1 \cdots \in (2^{\Sigma})^{\omega}$, a run of σ on $\mathcal{A}$ is an infinite sequence $q_0 q_1 \cdots \in Q^{\omega}$ of states where $(q_i, \sigma_i, q_{i+1}) \in \delta$ holds for all $i \geq 0$. A run is called accepting if it contains only finitely many rejecting states. $\mathcal{A}$ accepts a word σ if all runs of σ on $\mathcal{A}$ are accepting. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all accepted words. An LTL specification φ can be translated into an equivalent universal co-Büchi automaton $\mathcal{A}_{\varphi}$ with a single exponential blow up [14].

Decomposition. A decomposition is a partitioning of the system into components. A component p is defined by its input variables $\mathit{inp}(p) \subseteq (\mathit{inp} \cup \mathit{out})$ and output variables $\mathit{out}(p) \subseteq \mathit{out}$ with $\mathit{inp}(p) \cap \mathit{out}(p) = \emptyset$, where inp and out are the input and output variables of the system and $V = \mathit{inp} \cup \mathit{out}$. The output variables of components are pairwise disjoint and their union is equivalent to out. The *implementation order* defines the communication interface between the components. It assigns a rank $\mathit{rank}_{\mathit{impl}}(p)$ to every component p. If $\mathit{rank}_{\mathit{impl}}(p) < \mathit{rank}_{\mathit{impl}}(p')$, then p' sees the valuations of the variables in $\mathit{inp}(p') \cap \mathit{out}(p)$ one step in advance, i.e., it is able to directly react to them, modeling knowledge about these variables in the whole system. The implementation order is not necessarily total.

Strategies. A strategy is a function $s : (2^{inp(p)})^* \to 2^{out(p)}$ that maps a history of inputs of a component p to outputs. We model strategies as Moore machines $\mathcal{T} = (T, t_0, \tau, o)$ with a finite set of states T, an initial state t_0, a transition function $\tau : T \times 2^{inp(p)} \to T$, and an output function $o : T \to 2^{out(p)}$ that is is independent of the input. Given an input sequence $\gamma = \gamma_0\gamma_1 \ldots \in (2^{V \setminus out(p)})^\omega$, $\mathcal{T}$ produces a path $\pi = (t_0, \gamma_0 \cup o(t_0, \gamma_0))(t_1, \gamma_1 \cup o(t_1, \gamma_1)) \ldots \in (T \times 2^V)^\omega$, where $\tau(t_j, \boldsymbol{i}_j) = t_{j+1}$. The projection of a path to the variables is called trace. The trace produced by $\mathcal{T}$ on γ is called the computation of strategy s represented by $\mathcal{T}$ on γ, denoted $comp(s, \gamma)$. A strategy s is *winning* for φ if $comp(s, \gamma) \models \varphi$ for all $\gamma \in (2^{inp})^\omega$. A strategy s is *dominated* by a strategy t for φ if for all $\gamma \in (2^{V \setminus out(p)})^\omega$ with $comp(s, \gamma) \models \varphi$, $comp(t, \gamma) \models \varphi$ holds as well. A strategy is *dominant* if it dominates every other strategy. A specification φ is called *admissible* if there exists a dominant strategy for φ.

Bounded Synthesis. Given a specification, synthesis derives an implementation that is correct by construction. Bounded synthesis [10] additionally requires a bound $b \in \mathbb{N}$ on the size of the implementation as input. It produces size-optimal strategies. The search for a strategy satisfying the specification is encoded into a constraint system. If it is unsatisfiable, then the specification is unrealizable for the given size bound. Otherwise, the solution defines a winning strategy. There exist SMT [10] as well as SAT, QBF, and DQBF [5] encodings.

4 Synthesis of Dominant Strategies

In our incremental synthesis approach, we seek for dominant strategies, rather than for winning ones. To synthesize dominant strategies, we construct a universal co-Büchi automaton $\mathcal{A}_\varphi^{dom}$ for a specification φ that accepts exactly the computations of dominant strategies following the ideas in [3,4]. As for the universal co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, the size of $\mathcal{A}_\varphi^{dom}$ is exponential in the length of φ [4]. In bounded synthesis, the automaton $\mathcal{A}_\varphi^{dom}$ is then used instead of $\mathcal{A}_\varphi$ to derive dominant strategies.

Since we synthesize independent components compositionally, dominance of the parallel composition of dominant strategies is crucial for both soundness and completeness. Yet, in contrast to winning strategies, the parallel composition of dominant strategies is not guaranteed to be dominant in general. Consider a system with components p_1 and p_2 that send each other messages m_1 and m_2, and the specification $\varphi = \diamondsuit m_1 \land \diamondsuit m_2$. For p_1, it is dominant to wait for m_2 before sending m_1 since this strategy only violates $\diamondsuit m_1$ if $\diamondsuit m_2$ is violated as well. Analogously, it is dominant for p_2 to wait for m_1 before sending m_2. The parallel composition of these strategies, however, never sends any message. It violates φ in every situation while there are strategies that are winning for φ. Nevertheless, dominant strategies are compositional for safety specifications:

Theorem 1 ([4]). *Let φ be a safety property and let s_1 and s_2 be strategies for components p_1 and p_2. If s_1 is dominant for φ and p_1 and s_2 is dominant for φ and p_2, then the parallel composition $s_1 \,||\, s_2$ is dominant for φ and $p_1 \,||\, p_2$.*

Algorithm 1: Incremental Synthesis

```
Input: specification $\varphi$, array $C$ of arrays of $k$ components ordered by $<_{syn}$
Output: strategies $s_1, \dots, s_k$ such that $s_1 \,||\, \dots \,||\, s_k$ is dominant for $\varphi$
array[k] strategies
strategy assumedStrategies
for $i = 1$ to $C.length()$ by 1 do
    strategy addForLayer
    for $j = 1$ to $C[i].length()$ by 1 do
        synthesize strategy $s$ for $C[i][j]$ such that (assumedStrategies || $s$) is
          dominant for $\varphi$
        int component = $C[i][j].getLabel()$
        strategies[component] = $s$
        addForLayer = addForLayer || $s$
    assumedStrategies = assumedStrategies || addForLayer
```

We extend this result to specifications where only a single component affects the liveness part. Intuitively, then a violation of the liveness part can always be lead back to the single component affecting it, contradicting the assumption that its strategy is dominant.

Theorem 2. *Let φ be a property where only output variables of component p_1 affect the liveness part of φ, and let s_1 and s_2 be two strategies for components p_1 and p_2, respectively. If s_1 is dominant for φ and p_1 and s_2 is dominant for φ and p_2, then the parallel composition $s_1 \,||\, s_2$ is dominant for φ and $p_1 \,||\, p_2$.*

To lift compositional synthesis to real-world settings where strategies have to rely on the fact that other components will not maliciously violate the specification, we circumvent the need for the existence dominant strategies for every component in the following sections: We model the assumption that other components behave in a dominant fashion by synthesizing strategies incrementally.

5 Incremental Synthesis

In this section, we introduce a synthesis algorithm based on dominant strategies, where, in contrast to compositional synthesis, the components are not necessarily synthesized independently but one after another. The strategies that are already synthesized provide further information to the one under consideration.

For the self-driving car from Section 2, for instance, there is no dominant strategy for the acceleration unit. However, when provided with a dominant gearing strategy, there is even a winning strategy for the acceleration unit. Therefore, synthesizing strategies for the components incrementally, rather than compositionally, allows us to synthesize a strategy for the self-driving car.

The incremental synthesis algorithm is described in Algorithm 1. Besides a specification φ, it expects an array of arrays of components that are ordered by the *synthesis order* $<_{syn}$ as input. The synthesis order assigns a rank $rank_{syn}(p_i)$

to every component p_i. Strategies for components with lower ranks are synthesized before strategies for components with higher ranks. Strategies for components with the same rank are synthesized compositionally. Thus, to guarantee soundness, the synthesis order has to ensure that either φ is a safety property, or that at most one of these components affects the liveness part of φ.

First, we synthesize dominant strategies $s_1, \dots, s_i$ for the components with the lowest rank in the synthesis order. Then, we synthesize dominant strategies $s_{i+1}, \dots, s_j$ for the components with the next rank *under the assumption* of the parallel composition of $s_1, \dots, s_i$, denoted $s_1 \,||\, \dots \,||\, s_i$. Particularly, we seek for strategies such that $s_1 \,||\, \dots \,||\, s_i \,||\, s_{i+\ell}$ is dominant for φ and $p_1 \,||\, \dots \,||\, p_i \,||\, p_{i+\ell}$, where $1 \leq \ell \leq j - i$. We continue until strategies for all components have been synthesized. The soundness follows directly from the construction of the algorithm as well as Theorems 1 and 2.

Theorem 3 (Soundness). *Let φ be a specification and let $s_1, \dots, s_k$ be the strategies produced by the incremental synthesis algorithm. Then $s_1 \,||\, \dots \,||\, s_k$ is dominant for φ. If φ is realizable, then $s_1 \,||\, \dots \,||\, s_k$ is winning.*

The success of incremental synthesis relies heavily on the choice of components. Clearly, it succeeds if compositional synthesis does. Otherwise, the synthesis order has to guarantee admissibility of every component when provided with the strategies of components with a lower rank. In this regard, it is crucial that the parallel composition of the components with the same rank is dominant. Thus, we introduce techniques for component selection inducing a synthesis order that ensure completeness of incremental synthesis in the following sections.

6 Semantic Component Selection

The component selection algorithm introduced in this section is based on dependencies between the output variables of the system. It directly induces a synthesis order ensuring completeness of incremental synthesis.

We require specifications to be of the form $(\varphi_1^A \wedge \dots \wedge \varphi_n^A) \rightarrow (\varphi_1^G \wedge \dots \wedge \varphi_m^G)$, where the conjuncts are conjunction-free in negation normal form. When seeking for dominant strategies, assumptions can be treated as conjuncts as long as the system is not able to satisfy the specification by violating the assumptions. Since it is a modeling flaw if the assumptions can be violated by the system, we assume specifications to be of the form $(\varphi_1^A \wedge \dots \wedge \varphi_n^A) \wedge (\varphi_1^G \wedge \dots \wedge \varphi_m^G)$ in the following.

First, we introduce an algorithm for component selection that ensures completeness of incremental synthesis in the absence of input variables. Afterwards, we extend it to achieve completeness in general. The algorithm identifies equivalence classes of variables based on dependencies between them. These equivalence classes then build the components. Intuitively, a variable u depends on the current or future valuation of a variable v if changing the valuation of u yields a violation of the specification φ that can be fixed by changing the valuation of v at the same point in time or at a strictly later point in time, respectively. The change of the valuation of v needs to be necessary for the satisfaction of φ in the sense that not changing it would not yield a satisfaction of φ.

Definition 1 (Minimal Satisfying Changeset). *Let φ be a specification, let $\gamma \in (2^{inp})^\omega$, $\pi \in (2^{out})^\omega$ be sequences such that $\gamma \cup \pi \not\models \varphi$, let $u \in out$ and let i be a position. For sets $P \subseteq out \setminus \{u\}$, $F \subseteq out$, let $\Pi^{P,F}$ be the set of output sequences $\pi^{P,F} \in (2^{out})^\omega$ such that $\pi^{P,F}_j = \pi_j$ for all $j < i$ and*

- *$\forall v \in P.\ v \in \pi^{P,F}_i \leftrightarrow v \notin \pi_i$ and $\forall v \in V \setminus P.\ v \in \pi^{P,F}_i \leftrightarrow v \in \pi_i$, and*
- *$\forall v \in F.\ \exists j > i.\ v \in \pi^{P,F}_j \leftrightarrow v \notin \pi_j$ and $\forall v \in V \setminus F.\ \forall j > i.\ v \in \pi^{P,F}_j \leftrightarrow v \in \pi_j$.*

If there is a sequence $\pi^{P,F} \in \Pi^{P,F}$, such that $\gamma \cup \pi^{P,F} \models \varphi$ and for all $P' \subset P$, $F' \subset F$, we have $\gamma \cup \pi^{P',F'} \not\models \varphi$ for all $\pi^{P',F'} \in \Pi^{P',F'}$, then (P, F) is called minimal satisfying changeset *with respect to φ, γ, π, i.*

Definition 2 (Semantic Dependencies). *Let φ be a specification, let $u \in out$. Let $\eta, \eta' \in (2^V)^*$ be sequences of length $i + 1$ such that $u \in \eta'_i \leftrightarrow u \notin \eta_i$, $\forall v \in V \setminus \{u\}.\ v \in \eta'_i \leftrightarrow v \in \eta_i$, and $\forall j < i.\ \eta'_j = \eta_j$. If there are $\gamma \in (2^{inp})^\omega$, ${}_\gamma\pi \in (2^{out})^\omega$ with $\gamma_0 \dots \gamma_i = \eta \cap inp$, ${}_\gamma\pi_0 \dots {}_\gamma\pi_i = \eta \cap out$, and $\gamma \cup {}_\gamma\pi \models \varphi$, then*

- *u depends on (P, F) for $P \subseteq out \setminus \{u\}$, $F \subseteq out$ if there is ${}_\gamma\pi' \in (2^{out})^\omega$ with ${}_\gamma\pi'_0 \dots {}_\gamma\pi'_i = \eta' \cap out$ and ${}_\gamma\pi_j = {}_\gamma\pi'_j$ for all $j > i$ such that $\gamma \cup {}_\gamma\pi' \not\models \varphi$ and (P, F) is a minimal satisfying changeset w.r.t. φ, γ, ${}_\gamma\pi'$, i. We say that u* depends semantically on the current or future valuation *of v, if there are P, F such that u depends on (P, F) and $v \in P$ or $v \in F$, respectively.*
- *u* depends on the input, *if for all ${}_\gamma\pi'' \in (2^{out})^\omega$ with ${}_\gamma\pi''_0 \dots {}_\gamma\pi''_i = \eta' \cap out$, we have $\gamma \cup {}_\gamma\pi'' \not\models \varphi$, while there are $\gamma' \in (2^{inp})^\omega$, ${}_{\gamma'}\pi'' \in (2^{out})^\omega$ with $\gamma'_0 \dots \gamma'_i = \eta \cap inp$ and ${}_{\gamma'}\pi''_0 \dots {}_{\gamma'}\pi''_i = \eta' \cap out$ such that $\gamma' \cup {}_{\gamma'}\pi'' \models \varphi$.*

The specification of the self-driving car induces, for instance, a present dependency from *acc* to *dec*: Let $\gamma = \emptyset^\omega$, $\eta = \{gear_1, dec\}$, $\eta' = \{gear_1, dec, acc\}$. For ${}_\gamma\pi = \{gear_1, dec\}\{gear_2\}^\omega$, $\gamma \cup {}_\gamma\pi$ clearly satisfies φ_{car}. In contrast, for ${}_\gamma\pi' = \{gear_1, dec, acc\}\{gear_2\}^\omega$, $\gamma \cup {}_\gamma\pi' \not\models \varphi_{car}$ since mutual exclusion of *acc* and *dec* is violated. For $P = \{dec\}$, $F = \emptyset$, (P, F) is a minimal satisfying changeset w.r.t. φ, γ, ${}_\gamma\pi'$, i. Thus, *acc* depends on the current valuation of *dec*.

If a variable u depends on the future valuation of some variable v, a strategy for u most likely has to predict the future, preventing the existence of a dominant strategy for u. In our setting, strategies cannot react directly to an input. Thus, present dependencies may prevent admissibility as well. Yet, the implementation order resolves a present dependency from u to v if $rank_{impl}(v) < rank_{impl}(u)$: Then, the valuation of v is known to u one step in advance and thus a strategy for u does not have to predict the future. Hence, if u neither depends on the input, nor on the future valuation of some $v \in out$, nor on its current valuation if $rank_{impl}(u) \leq rank_{impl}(v)$, then the specification is admissible for u.

To show this formally, we construct a dominant strategy for u. It maximizes the set of input sequences for which there is an output sequence that satisfies the specification. In general, this strategy is not dominant since these output sequences may not be computable by a strategy. Yet, this can only be the case if a strategy needs to predict the valuations of variables outside its control and this need is exactly what is captured by semantic present and future dependencies.

Theorem 4. *Let φ be a specification and let $O \subseteq out$. If for all $u \in O$, u neither depends semantically on the future valuation of v, nor on the current valuation of v if $rank_{impl}(u) \leq rank_{impl}(v)$ for all $v \in V \setminus O$, nor on the input, then φ is admissible for the component p with $out(p) = O$.*

We build a dependency graph in order to identify the components of the system. The vertices represent the variables and edges denote semantic dependencies between them. Formally, the *Semantic Dependency Graph* $\mathcal{D}^{sem}_{\varphi} = (V_{\varphi}, E^{sem}_{\varphi})$ of φ is given by $V_{\varphi} = V$ and $E^{sem}_{\varphi} = E^{sem}_{\varphi,p} \cup E^{sem}_{\varphi,f} \cup E^{sem}_{\varphi,i}$, where $(u, v) \in E^{sem}_{\varphi,p}$ if u depends on the current valuation of $v \in out$, $(u, v) \in E^{sem}_{\varphi,f}$ if u depends on the future valuation of $v \in out$, and $(u, v) \in E^{sem}_{\varphi,i}$ if u depends on $v \in inp$.

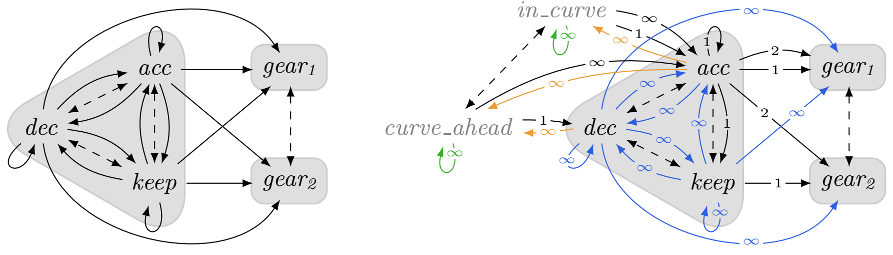
To identify the components, we proceed in three steps. First, we eliminate vertices representing input variables since they are not part of the components. Second, we resolve present dependencies. Since future dependencies subsume present ones, we remove (u, v) from $E^{sem}_{\varphi,p}$ if $(u, v) \in E^{sem}_{\varphi,f}$. Then, we resolve present dependencies by refining the implementation order: If $(u, v) \in E^{sem}_{\varphi,p}$, we add $rank_{impl}(v) < rank_{impl}(u)$ and remove (u, v) from $E^{sem}_{\varphi,p}$. This is only possible if the implementation order does not become contradictory. In particular, at most one present dependency between u and v can be resolved in this way. Third, we identify the strongly connected components $\mathcal{C} := \{C_1, \ldots, C_k\}$ of $\mathcal{D}^{sem}_{\varphi}$. They define the decomposition of the system: We obtain k components $p_1, \ldots, p_k$ with $out(p_i) = C_i$ for $1 \leq i \leq k$. Thus, the number of strongly connected components should be maximized when resolving present dependencies in step two.

The dependency graph induces the *synthesis order*: Let $\mathcal{C}^i \subseteq \mathcal{C}$ be the set of strongly connected components that do not have any direct predecessor when removing $\mathcal{C}^0 \cup \cdots \cup \mathcal{C}^{i-1}$ from $\mathcal{D}^{sem}_{\varphi}$. For all $C_n \in \mathcal{C}^0$, $rank_{syn}(p_n) = 1$. For $C_n \in \mathcal{C}^i$, $C_m \in \mathcal{C}^j$, $rank_{syn}(p_n) < rank_{syn}(p_m)$ if $i > j$ and $rank_{syn}(p_n) > rank_{syn}(p_m)$ if $i < j$. If $i = j$, $rank_{syn}(p_n) = rank_{syn}(p_m)$ if φ is a safety property or only one of the components affects the liveness part of φ. Otherwise, choose an ordering, i.e., either $rank_{syn}(p_n) < rank_{syn}(p_m)$ or $rank_{syn}(p_m) < rank_{syn}(p_n)$.

For the specification of the self-driving car, we obtain the semantic dependency graph shown in Figure 1a. It induces three components p_1, p_2, p_3 with $out(p_1) = \{gear_1\}$, $out(p_2) = \{gear_2\}$, and $out(p_3) = \{acc, dec, keep\}$. When adding $rank_{impl}(gear_2) < rank_{impl}(gear_1)$ to the implementation order, we obtain $rank_{syn}(p_1) < rank_{syn}(p_2) < rank_{syn}(p_3)$ and thus $p_1 <_{syn} p_2 <_{syn} p_3$.

Incremental synthesis with the semantic component selection algorithm is complete for specifications that do not contain dependencies to input variables: By construction, a component $p \in \mathcal{C}^0$ has no unresolved semantic dependencies to variables outside of p. Thus, by Theorem 4, φ is admissible. Moreover, by the incremental synthesis algorithm as well as Theorems 1 and 2, for every component $p \in \mathcal{C}^i$, the parallel composition of the strategies of components p' with $rank_{syn}(p') < rank_{syn}(p)$ is dominant. Thus, by construction, there is a dominant strategy for $\mathcal{C}^0 \cup \cdots \cup \mathcal{C}^i$ as well.

Lemma 1. *Let φ be a specification. If for all $u \in out$, u does not depend semantically on the input, then incremental synthesis yields strategies for all components and the synthesis order induced by the component selection algorithm.*



(a) Semantic Dependency Graph (b) Syntactic Dependency Graph

Fig. 1: Semantic and Syntactic Dependency Graphs for the self-driving car. Dashed edges denote present dependencies, solid ones future dependencies. Gray boxes denote induced components. In (b), blue edges are obtained by transitivity, orange ones by derivation, and green ones by transitivity after derivation. For the sake of readability, not all transitive and derived edges are displayed.

Since semantic dependencies to input variables cannot be resolved, admissibility is not guaranteed in general. Yet, if the specification is realizable, admissibility of completely independent components follows: If p does not depend on the input, admissibility of φ follows directly with Lemma 1. Otherwise, φ can only be non-admissible for p if a strategy has to predict the valuation of an input variable. Since p is completely independent of other components, a different valuation of an output variable outside of p cannot affect the need to predict input variables. But then a strategy for the whole system has to predict inputs as well, yielding a contradiction.

Theorem 5. *Let φ be a specification, let p be a component such that for all p', $rank_{syn}(p') \leq rank_{syn}(p)$, and for all $u \in out(p)$, u neither depends semantically on the future valuation of $v \in out \setminus out(p)$, nor on its current valuation if $rank_{impl}(u) \leq rank_{impl}(v)$. If φ is realizable, then φ is admissible for p.*

Thus, when encountering a component for which φ is not admissible in incremental synthesis, we can directly deduce non-realizability of φ if there is no component with a higher rank in the synthesis order. Yet, this does not hold in general. Consider $\varphi = a \vee ((\bigcirc b) \leftrightarrow (\bigcirc\bigcirc i))$, where i is an input variable and both a and b are output variables. Since a depends on b while b does not depend on a, a strategy for b has to be synthesized first. Yet, there is no dominant strategy for b since it has to predict the future valuation of i, while there is a dominant strategy for the whole system, namely the one that sets a in the first step.

Thus, we combine a component for which φ is not admissible with a direct successor in the synthesis order until either φ is admissible or only a single component is left. With this extension, the completeness of incremental synthesis follows directly from Lemma 1 and Theorem 5.

Theorem 6 (Completeness). *Let φ be a specification. If φ is realizable, incremental synthesis yields strategies for all components and the synthesis order induced by the extended semantic component selection algorithm.*

7 Syntactic Analysis

While analyzing semantic dependencies for component selection ensures completeness of incremental synthesis, computing the dependencies is hard. In particular, the semantic definition of dependencies is a hyperproperty [2], i.e., a property relating multiple execution traces, with quantifier alternation. To determine the present and future dependencies between variables more efficiently, we introduce a dependency definition based on the syntax of the LTL formula.

Definition 3 (Syntactic Dependencies). *Let φ be an LTL formula in negation normal form. Let $\mathcal{T}(\varphi)$ be the syntax tree of φ, where $\Box\Diamond$ is considered to be a separate operator. Let q be a node of $\mathcal{T}(\varphi)$ with child q', if q is a unary operator, and left child q' and right child q'', if q is a binary operator. We assign a set $D_q \in 2^{2^{V \times \mathbb{N} \times \mathbb{B}}}$ to each node q of $\mathcal{T}(\varphi)$ as follows:*

- *if q is a leaf, then $q = u \in V$ and $D_q = \{\{(u, 0, \mathit{false})\}\}$,*
- *if $q = \neg$, then $D_q = D_{q'}$,*
- *if $q = \wedge$, then $D_q = D_{q'} \cup D_{q''}$,*
- *if $q = \vee$, then $D_q = \bigcup_{M \in D_{q'}} \bigcup_{M' \in D_{q''}} \{M \cup M'\}$,*
- *if $q = \bigcirc$, then $D_q = \bigcup_{M \in D_{q'}} \{\{(u, x+1, y) \mid (u, x, y) \in M\}\}$,*
- *if $q = \Box$, then $D_q = D_{q'} \cup \bigcup_{M \in D_{q'}} \{\{(u, x, \mathit{true})\} \mid (u, x, y) \in M\}$,*
- *if $q = \Diamond$, then $D_q = D_{q'} \cup \left\{\bigcup_{M \in D_{q'}} \{(u, x, \mathit{true}), (u, x, \mathit{false}) \mid (u, x, y) \in M\}\right\}$*
- *if $q = \Box\Diamond$, then $D_q = \bigcup_{M \in D_{q'}} \{\{(u, x, \mathit{true})\} \mid (u, x, y) \in M\}$,*
- *if $q = \mathcal{U}$ or $q = \mathcal{W}$, then*

$$\begin{aligned} D_q = & \bigcup_{M \in D_{q'}} \bigcup_{M' \in D_{q''}} \{M \cup M'\} \\ & \cup \bigcup_{M \in D_{q'}} \bigcup_{M' \in D_{q''}} \bigcup_{(u,x,y) \in M} \{\{(u, x, \mathit{true})\} \cup M'\} \\ & \cup \left\{ \bigcup_{M' \in D_{q''}} \{(u, x, \mathit{true}), (u, x, \mathit{false}) \mid (u, x, y) \in M'\} \right\} \end{aligned}$$

Let q be the root node of $\mathcal{T}(\varphi)$ and let $(u, x, y), (v, x', y') \in M$ for some $M \in D_q$, $u, v \in V$, $x, x' \in \mathbb{N}$, and $y, y' \in \mathbb{B}$ with $(u, x, y) \neq (v, x', y')$. Then u depends syntactically on the current valuation of *v, if $u \neq v$ and either $y = y' = \mathit{false}$ and $x = x'$, or $y = \mathit{true}$ and $y' = \mathit{false}$ and $x \leq x'$, or $y = \mathit{false}$ and $y' = \mathit{true}$ and $x \geq x'$, or $y = y' = \mathit{true}$. Furthermore, u* depends syntactically on the future valuation of *v, if either $y' = \mathit{true}$, or $y' = \mathit{false}$ and $x < x'$. The* offset *of the future dependency is ∞ in the former case and $x' - x$ in the latter case.*

For (u, x, y), x denotes the number of $\bigcirc$-operators under which u occurs and y denotes whether u occurs under an unbounded temporal operator. Since the specification is in negation normal form, negation only occurs in front of variables and thus does not influence the dependencies. Disjunction introduces dependencies between the disjuncts ψ and ψ' since the satisfaction of ψ affects the need of satisfaction of ψ' and vice versa. A conjunct, however, has to be satisfied irrespective of other conjuncts and thus conjunction does not introduce dependencies. Analogously, $\diamondsuit\,\psi$ introduces future dependencies between the variables in ψ, while $\square\,\psi$ does not. Adding triples with both *true* and *false* is necessary for the $\diamondsuit$-operator in order to obtain future dependencies from a variable to itself also if ψ contains only a single variable, e.g., for $\diamondsuit\,u$. For $\psi\,\mathcal{U}\,\psi'$ and $\psi\mathcal{W}\psi'$, there are dependencies between ψ and ψ' as well as future dependencies between the variables in ψ' analogously to disjunction and the $\diamondsuit$-operator. Furthermore, there are future dependencies from ψ' to ψ since whether or not ψ is satisfied in the future affects the need of satisfaction of ψ' in the current step. The $\square\diamondsuit$-operator takes a special position. Although including $\diamondsuit$, changing the valuation of a variable at a single position does not yield a violation of $\square\diamondsuit\,\psi$ and thus there is no semantic dependency. Hence, $\square\diamondsuit\,\psi$ does not introduce syntactic dependencies between the variables in ψ either.

For the specification of the self-driving car from Section 2, we annotate, for instance, node q representing the $\square$-operator of the conjunct $\square\neg(\mathit{acc} \land \mathit{dec})$ with $D_q = \{\{(\mathit{acc}, 0, \mathit{false}), (\mathit{dec}, 0, \mathit{false})\}, \{(\mathit{acc}, 0, \mathit{true})\}, \{(\mathit{dec}, 0, \mathit{true})\}\}$, yielding a syntactic present dependency from *acc* to *dec* and vice versa. For the node q representing the $\square$-operator of $\square((\mathit{acc} \land \bigcirc\,\mathit{acc}) \rightarrow \bigcirc\bigcirc\,\mathit{gear_1})$, we obtain amongst others $\{(\mathit{acc}, 0, \mathit{false}), (\mathit{acc}, 1, \mathit{false}), (\mathit{gear_1}, 2, \mathit{false})\} \in D_q$, yielding future dependencies from *acc* to *acc* with offset 1 and to $\mathit{gear_1}$ with offsets 1 and 2.

As long as semantic dependencies do not range over several conjuncts, every semantic dependency is captured by a syntactic one as well: If there is a semantic dependency from u to v and if φ does not contain any conjunction, u and v occur in the same set $M \in D_q$, where q is the root node of $\mathcal{T}(\varphi)$, by construction. With structural induction on φ, it thus follows that every semantic dependency has a syntactic counterpart.

Lemma 2. *Let φ be an LTL formula in negation normal form that does not contain any conjunction. Let $u, v \in V$ be variables. If u depends semantically on the current or future valuation of v, then u depends syntactically on the current or future valuation of v, respectively, as well.*

Yet, the above definition of syntactic dependencies does not capture all semantic dependencies in general. Particularly, semantic dependencies ranging over several conjuncts cannot be detected. To capture all dependencies, we build the *syntactic dependency graph* analogously to the semantic one, additionally annotating future dependency edges with their offsets. We build the *transitive closure* over output variables: Let $u, v \in \mathit{out}$ and let there be $u_1, \ldots, u_j \in \mathit{out}$ for some $j \geq 1$ with $(u, u_1) \in E^{syn}_{\varphi}$, $(u_j, v) \in E^{syn}_{\varphi}$, and $(u_i, u_{i+1}) \in E^{syn}_{\varphi}$ for all $1 \leq i < j$. If all these edges are present dependency edges, then $(u, v) \in E^{syn}_{\varphi,p}$. Otherwise, $(u, v) \in E^{syn}_{\varphi,f}$. If there are connecting edges for u and v containing a future dependency cycle, the offset of the transitive edge is ∞. Otherwise, it is the sum of the offsets of the connecting edges. To capture the synergy of dependencies, let $u, v, w \in V$ be variables with $u, w \in \mathit{out}$ and $u \neq v$ or $u \neq w$. Let $(u, w) \in E^{syn}_{\varphi,f}$ with offset x and $(v, w) \in E^{syn}_{\varphi,f}$ with offset y. If $x \neq \infty$ and $y \neq \infty$, then, if

$x = y$, add (u, v) and (v, u) to $E^{syn}_{\varphi,p}$, and if $x < y$ or $x > y$, add (v, u) or (u, v) to $E^{syn}_{\varphi,f}$ with offset $y - x$ or $x - y$, respectively. If $x = \infty$, add both $(u, v), (v, u)$ to $E^{syn}_{\varphi,p}$ and $E^{syn}_{\varphi,f}$ with offset ∞. Build the transitive closure again.

The resulting syntactic dependency graph for the self-driving car is shown in Figure 1b. Unlike the semantic one, it contains outgoing dependencies from input variables. While such dependencies are not relevant for component selection and thus are not defined in the semantic algorithm, they are needed to derive dependencies *to* input variables with the syntactic technique.

After the derivation of further dependencies in the dependency graph, every semantic dependency has a syntactic counterpart, even if it ranges over several conjuncts. Intuitively, the derivation of a minimal satisfying changeset for a semantic dependency induces several separate semantic present and future dependencies that only affect single conjuncts of the specification. With Lemma 2, the claim follows by induction on the number of these separate dependencies.

Theorem 7. *Let φ be an LTL formula and let $u, v \in out$. If $(u, v) \in E^{sem}_{\varphi,p}$, then $(u, v) \in E^{syn}_{\varphi,p}$. If $(u, v) \in E^{sem}_{\varphi,f}$, then $(u, v) \in E^{syn}_{\varphi,f}$. If u depends semantically on the input, then there are variables $w \in out$, $w' \in inp$ such that $(w, w') \in E^{syn}_{\varphi}$.*

Thus, since semantic dependencies have a syntactic counterpart, completeness of incremental synthesis using syntactic dependency analysis for selecting components follows directly with Theorem 6. However, the syntactic analysis is a conservative overapproximation of the semantic dependencies. This can be easily seen when comparing the semantic and syntactic dependency graphs for the self-driving car shown in Figure 1. For instance, there is a syntactic future dependency from *acc* to *in_curve* while there is no such semantic dependency. In particular, the derivation rules are blamable for the overapproximation.

8 Specification Simplification

In this section, we identify conjuncts that are not relevant for the component p under consideration to reduce the size of the specification. In general, leaving out conjuncts is not sound since the missing conjuncts may invalidate admissibility of the specification [4]. However, non-admissible components cannot become admissible by leaving out conjuncts that do not refer to output variables of p:

Theorem 8 ([4]). *Let φ be an LTL formula over $V \setminus out(p)$ and let ψ be an LTL formula over V. If ψ is admissible, then $\varphi \land \psi$ is admissible as well.*

Yet, an admissible component may become non-admissible. For instance, consider the specification $\varphi = \square(a \leftrightarrow \bigcirc i) \land \square i$, where i is an input variable and a is an output variable. While always outputting a is a dominant strategy for φ, leaving out $\square i$ yields non-admissibility of φ since a dominant strategy for a needs to predict i. A conjunct that does not contain variables on which the component under consideration depends, however, can be eliminated since its satisfaction does not influence the admissibility of the specification for p:

Theorem 9. *Let φ be an LTL formula such that $\varphi = \psi \land \psi'$, where ψ is an LTL formula over $V' \subseteq V \setminus out(p)$ not containing assumption conjuncts and ψ' is an LTL formula over V. If for all $u \in out(p)$ and $v \in out \setminus out(p)$, u neither depends on the future valuation of v, nor on the present valuation of v if $rank_{impl}(u) \leq rank_{impl}(v)$, and if φ is realizable for the whole system, then ψ' is admissible for p if, and only if, φ is admissible for p.*

If ψ' is admissible, admissibility of φ follows since the truth value of ψ is solely determined by the input of p. Otherwise, a strategy for p has to predict the input. Since p is independent of all other components, φ can only be realizable if ψ restricts the input behavior, contradicting the assumption that it does not contain assumption conjuncts. This directly leads to the following observation:

Corollary 1. *Let $\varphi = \psi \land \psi'$ be an LTL formula inducing two components p, p' with $rank_{syn}(p) = rank_{syn}(p')$ for either the semantic or the syntactic technique, where ψ and ψ' range over $V \setminus out(p')$ and $V \setminus out(p)$, respectively. If φ is realizable, then there are winning strategies for p and p' for ψ and ψ', respectively.*

Moreover, in incremental synthesis the strategies of components with a lower rank in the synthesis order are provided to the component p under consideration. Hence, if these strategies are winning for a conjunct, it may be eliminated from the specification for p since its satisfaction is already guaranteed.

Theorem 10. *Let φ, ψ be LTL formulas over V. Let s' be the parallel composition of the strategies for the components p_i with $rank_{syn}(p_i) < rank_{syn}(p)$. If s' is winning for φ, then there is a strategy s such that $s' \,||\, s$ is dominant for ψ if, and only if, there is a strategy s such that $s' \,||\, s$ is dominant for $\varphi \land \psi$.*

9 Experimental Results

We implemented a prototype of the incremental synthesis algorithm. It expects an LTL specification as well as a decomposition of the system and a synthesis order as input. Our prototype extends the state-of-the-art synthesis tool BoSy [6] to the synthesis of dominant strategies. Furthermore, it converts the synthesized strategy from the Aiger-circuit produced by our extension of BoSy to an equivalent LTL formula that is added to the specification of the next component.

We compare our prototype to BoSy on four scalable benchmarks. The results are presented in Table 1. The first two benchmarks stem from the reactive synthesis competition (SYNTCOMP 2018) [11]. The latch is parameterized in the number of bits and the Generalized Buffer in the number of receivers. For the n-ary latch, both the semantic and the syntactic component selection algorithms identify n separate components, one for each bit of the latch. For the Generalized Buffer, both techniques identify two components, one for the communication with the senders and one for the communication with the receivers. After simplifying the specification using Theorem 9, we are able to synthesize separate winning strategies for the components for both benchmarks, making use of Corollary 1. The incremental synthesis approach clearly outperforms BoSy's

Table 1: Experimental results on scalable benchmarks. Reported is the parameter and the time in seconds. We used a machine with a 3.1 GHz Dual-Core Intel Core i5 processor and 16 GB of RAM, and a timeout of 60 minutes.

Benchmark	Parameter	BoSy	Incremental Synthesis
n-ary Latch	2	**2.61**	4.76
	3	**3.66**	6.58
	4	11.55	**8.74**
	5	TO	**10.98**
	...	...	...
	1104	TO	**3599.04**
Generalized Buffer	1	37.04	**5.08**
	2	TO	**6.21**
	3	TO	**66.03**
Sensors	2	**1.99**	6.08
	3	**2.31**	8.79
	4	**6.99**	11.73
	5	92.79	**16.99**
	6	TO	**43.50**
	7	TO	**2293.85**
Robot Fleet	2	**2.49**	6.25
	3	TO	**10.51**
	4	TO	**269.09**

classical bounded synthesis approach for the Generalized Buffer in all cases. For the n-ary latch, the advantage becomes clear from $n = 4$ on.

Furthermore, we consider a benchmark describing n sensors and a managing unit that requests and collects sensor data. The semantic component selection technique identifies n separate components for the sensors as well as a component for the managing unit that depends on the other components. For this decomposition, the incremental synthesis approach outperforms BoSy for $n \geq 5$. The syntactic technique, however, does not identify the separability of the sensors from the managing unit due to the overapproximation in the derivation rules.

Lastly, we consider a benchmark describing a fleet of n robots that must not collide with a further robot crossing their way. Both the semantic and the syntactic technique identify n separate components for the robots in the fleet as well as a component for the further robot depending on the former components. Our prototype outperforms BoSy from $n \geq 3$ on. It still terminates in less than 5 minutes when BoSy is not able to synthesize a strategy within 60 minutes.

10 Conclusions

We have presented an incremental synthesis algorithm that reduces the complexity of synthesis by decomposing large systems. Furthermore, it is, unlike compositional approaches, applicable if the components depend on the strategies of other components. We have introduced two techniques to select the components, one based on a semantic dependency analysis of the output variables and one based on a syntactic analysis of the specification. Both induce a synthesis order that guarantees soundness and completeness of incremental synthesis. Moreover, we have presented rules for reducing the size of the specification for the components. We have implemented a prototype of the algorithm and compared it to a state-of-the-art synthesis tool. Our experiments clearly demonstrates the advantage of incremental synthesis over classical synthesis for large systems. The prototype uses a bounded synthesis approach. However, the incremental synthesis algorithm applies to other synthesis approaches, e.g., explicit approaches as implemented in the state-of-the-art tool Strix [15], as well if they are extended with the possibility of synthesizing dominant strategies.

References

1. Baier, C., Klein, J., Klüppelholz, S.: A Compositional Framework for Controller Synthesis. In: Proc. of CONCUR (2011)
2. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6) (2010)
3. Damm, W., Finkbeiner, B.: Does It Pay to Extend the Perimeter of a World Model? In: Proc. of FM (2011)
4. Damm, W., Finkbeiner, B.: Automatic Compositional Synthesis of Distributed Systems. In: Proc. of FM (2014)
5. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of Bounded Synthesis. In: Proc. of TACAS (2017)
6. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An Experimentation Framework for Bounded Synthesis. In: Proc. of CAV (2017)
7. Filiot, E., Jin, N., Raskin, J.: Compositional Algorithms for LTL Synthesis. In: Bouajjani, A., Chin, W. (eds.) Proc. of ATVA (2010)
8. Finkbeiner, B., Passing, N.: Dependency-based Compositional Synthesis (Full Version). CoRR **abs/2007.06941** (2020)
9. Finkbeiner, B., Schewe, S.: Semi-Automatic Distributed Synthesis. In: Proc. of ATVA (2005)
10. Finkbeiner, B., Schewe, S.: Bounded Synthesis. STTT (2013)
11. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results. CoRR **abs/1904.07736** (2019)
12. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Proc. of TACAS (2009)
13. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless Compositional Synthesis. In: Proc. of CAV (2006)
14. Kupferman, O., Vardi, M.Y.: Safraless Decision Procedures. In: Proc. of FOCS (2005)
15. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit Reactive Synthesis Strikes Back! In: Proceeding of CAV (2018)
16. de Roever, W.P., Langmaack, H., Pnueli, A. (eds.): Compositionality: The Significant Difference, COMPOS'97, LNCS, vol. 1536. Springer (1998)