Automatic Compositional Synthesis of Distributed Systems

Werner Damm[1] and Bernd Finkbeiner[2]

[1] Carl von Ossietzky Universität Oldenburg
[2] Universität des Saarlandes

Abstract. Given the recent advances in synthesizing finite-state controllers from temporal logic specifications, the natural next goal is to synthesize more complex systems that consist of multiple distributed processes. The synthesis of distributed systems is, however, a hard and, in many cases, undecidable problem. In this paper, we investigate the synthesis problem for specifications that admit dominant strategies, i.e., strategies that perform at least as well as the best alternative strategy, although they do not necessarily win the game. We show that for such specifications, distributed systems can be synthesized compositionally, considering one process at a time. The compositional approach has dramatically better complexity and is uniformly applicable to all system architectures.

1 Introduction

Synthesis, the automatic translation of specifications into implementations, holds the promise to revolutionize the development of complex systems. While the problem has been studied for a long time (the original formulation is attributed to Alonzo Church [4]), recent years seem to have achieved the phase transition to practical tools and realistic applications, such as the automatic synthesis of the AMBA bus protocol [1]. Tools like Acacia+ [3], Ratsy [2], and Unbeast [6] automatically translate a specification given in linear-time temporal logic into finite-state machines that guarantee that the specification holds for all possible inputs from the system's environment. Given the success of obtaining such finite-state controllers, the natural next step would be to synthesize more complex systems, consisting of multiple distributed processes. However, none of the currently available tools is capable of synthesizing systems with as many as two processes. This is unfortunate, because a separation into multiple processes is not only necessary to obtain well-structured and humanly understandable implementations, but is in fact often a non-negotiable design constraint: for example, the synchronization between different ECUs in a car involves explicit and time-consuming bus communication; approximating the network of ECUs with a single process therefore usually produces unimplementable solutions.

The lack of tools for the synthesis of distributed systems is no accident. For most system architectures, the distributed synthesis problem is undecidable [14], and for system architectures where the problem is decidable, such as pipelines, the complexity has been shown to be non-elementary in the number of processes. Experience with similar problems with non-elementary complexity, such as WS1S satisfiability (implemented

in Mona [10]), suggests, however, that these results do not necessarily mean that the synthesis of distributed systems is generally impossible. The specifications in the typical hardness arguments use the incomplete informedness of the processes to force the processes into specific complex behaviors. For example, in the undecidability proof due to Pnueli and Rosner, the specification forces the processes to simulate a Turing machine. The question arises if such specifications are of practical interest in the development of finite-state controllers. Can we obtain better complexity results if we restrict the specifications to a "reasonable" subset?

The key idea to reduce the complexity is to work *compositionally*. Compositionality is a classic concept in programming languages and verification where one ensures that the results obtained for a process also hold for the larger system [15]; in the case of synthesis, we want to ensure that the implementations found for individual processes can be used to realize the larger multi-process system. Unfortunately, synthesis does not lend itself easily to a compositional approach. In game-theoretic terms, synthesis looks for *winning strategies*, i.e., strategies that ensure the satisfaction of the specification under all circumstances. While the notion of *winning* is, in principle, compositional (if each process guarantees a property no matter what the other processes do, then clearly the system will guarantee the property as well), winning is too strong as a process requirement, because properties can rarely be guaranteed by one process alone. Typically, there exist input sequences that would prevent the process from satisfying the property, and the processes in the environment cooperate in the sense that they do not produce those sequences.

In this paper, we develop a synthesis technique for distributed systems that is based on a weaker notion than winning: A strategy is *dominant* if it performs, in any situation, at least as well as the best alternative strategy. Unlike winning strategies, dominant strategies are allowed to lose the game — as long as no other strategy would have won the game in the same situation. In a distributed system, a dominant strategy requires only a *best effort* – ensure the specification if you can – rather than a comprehensive *guarantee* that the specification is satisfied. It turns out that, just like winning, dominance is also a compositional notion. However, it is much more realistic to expect a process to have a dominant strategy than it is to have a winning strategy. In cases where the environment of the process behaves unreasonably, i.e., where it is made impossible for the process to satisfy its specification, we no longer require the process to satisfy the specification.

We call a specification that has a dominant strategy *admissible*. Intuitively, a specification is admissible as long as we do not require a process to "guess" variables it cannot see or to "predict" future inputs. Predicting future inputs is, of course, impossible; at the same time, it is easy to choose, in retrospect for a specific sequence of inputs, an alternative strategy that would have guessed correctly. Consider, for example, the LTL specification $\varphi = (\bigcirc a) \leftrightarrow b$, where a is an input variable and b is an output variable. By itself, φ is not admissible. Every specification can, however, be strengthened into an admissible specification. For example, $\varphi \wedge (\square b)$ is admissible.

As we show in the paper, there is a fundamental connection between admissibility and compositionality: a process has a dominant strategy if and only if there exists a unique weakest environment assumption that would guarantee that the process can

ensure the satisfaction of the specification. We first exploit this connection in an *incremental* synthesis algorithm: considering one process at a time, we compute the dominant strategy and the unique environment assumption. For the remaining processes, we replace the specification with the new assumption.

We then show that, for safety properties, true *compositionality* can be obtained by synthesizing each process *in isolation*. Even without considering the environment assumptions of the partner processes, the composition of the dominant strategies for two subarchitectures is guaranteed to result in a dominant strategy for the composite architecture.

Unfortunately, this property does not hold for liveness properties; the problem is that each process may have a dominant strategy that waits for the other process to make the first step. If such strategies are combined, they wait forever. We address this problem with a new notion of dominance, which we call *bounded dominance*. Intuitively, bounded dominance compares the number of steps that a strategy takes to satisfy a liveness objective with a (constant) bound. The dominant strategy must meet the bound whenever some alternative strategy would meet the bound. The composition of two strategies that are dominant for some bound is again dominant for the same bound.

Finally, we describe how to combine incremental and compositional synthesis, and how to localize the analysis based on an automatic decomposition of the specification into subsets of relevant properties for each process.

2 Synthesis of Distributed Systems

We are interested in synthesizing a distributed system for a given system architecture A and an LTL formula φ. A solution to the synthesis problem is a set of finite-state strategies $\{s_p \mid p \in P\}$, one for each process in the architecture, such that the joint behavior satisfies φ.

Architectures. An *architecture* A is a tuple (P, V, inp, out), where P is a set of system processes, V is a set of (Boolean) variables, and $inp, out : P \to 2^V$ are two functions that map each process to a set of input and output variables, respectively. For each process p, the inputs and outputs are disjoint, $inp(p) \cap out(p) = \emptyset$, and for two different processes $p \neq q$, the output variables are disjoint: $out(p) \cap out(q) = \emptyset$. We denote the set of visible variables of process p with $V(p) = inp(p) \cup out(p)$. If P is singleton, we call the architecture *single-process*; if P contains at least two processes, we call the architecture *distributed*.

For two architectures $A_1 = (P_1, V, inp_1, out_1)$ and $A_2 = (P_2, V, inp_2, out_2)$ with the same variables, but disjoint sets of processes, $P_1 \cap P_2 = \emptyset$, we define the parallel composition as the architecture $A_1 \| A_2 = (P_1 \cup P_2, V, p \mapsto$ if $p \in P_1$ then $inp_1(p)$ else $inp_2(p), p \mapsto$ if $p \in P_1$ then $out_1(p)$ else $out_2(p))$.

Implementations. An *implementation* of an architecture consists of strategies $S = \{s_p \mid p \in P\}$ for the system processes. A system process $p \in P$ is implemented by a *strategy*, i.e., a function $s_p : (2^{inp(p)})^* \to 2^{out(p)}$ that maps histories of inputs to outputs. A strategy is *finite-state* if it can be represented by a finite-state *transducer* $(Q, q_0, \delta : Q \times 2^{inp(p)} \to$

$Q, \gamma : Q \to 2^{out(p)}$), with a finite set of states Q, an initial state q_0, a transition function δ and an output function γ.

The parallel composition $s_p \| s_q$ of the strategies of two processes $p, q \in P$ is a function $s_{p\|q} : (2^I)^* \to 2^O$ that maps histories of the remaining inputs $I = (inp(p) \cup inp(q)) \setminus (out(p) \cup out(q))$ to the union $O = out(p) \cup out(q)$ of the outputs: $s_{p\|q}(\sigma) = s_p(\alpha_p(\sigma)) \cup s_q(\alpha_q(\sigma))$, where $\alpha_p(\epsilon) = \epsilon$ and $\alpha_p(\upsilon_0\upsilon_1 \dots \upsilon_k) = ((\upsilon_0 \cup s_q(\epsilon)) \cap inp(p))((\upsilon_1 \cup s_q(\alpha_q(\upsilon_0))) \cap inp(p)) \dots ((\upsilon_k \cup s_q(\alpha_q(\upsilon_1\upsilon_2 \dots \upsilon_{k-1}))) \cap inp(p))$, and, analogously, $\alpha_q(\epsilon) = \epsilon$ and $\alpha_q(\upsilon_0\upsilon_1 \dots \upsilon_k) = ((\upsilon_0 \cup s_p(\epsilon)) \cap inp(q))((\upsilon_1 \cup s_p(\alpha_p(\upsilon_0))) \cap inp(q)) \dots ((\upsilon_k \cup s_p(\alpha_p(\upsilon_1\upsilon_2 \dots \upsilon_{k-1}))) \cap inp(q))$.

A *computation* is an infinite sequence of variable valuations. For a sequence $\gamma = \upsilon_1\upsilon_2 \dots \in (2^{V \setminus out(p)})^\omega$ of valuations of the variables outside the control of a process p, the computation resulting from s is denoted by $comp(s, \gamma) = (s(\epsilon) \cup \upsilon_1)\,(s(\upsilon_1 \cap inp(p)) \cup \upsilon_2)\,(s(\upsilon_1 \cap inp(p)\upsilon_2 \cap inp(p)) \cup \upsilon_3) \dots$.

Specification. We use ω-regular languages, which we also call *properties*, to specify system behaviors. For a computation σ and an ω-regular language φ, we also write $\sigma \models \varphi$ if $\sigma \in \varphi$. To define ω-regular languages, we use automata or LTL formulas.

A strategy $s_p : (2^I)^* \to 2^O$ is *winning* for a property φ, denoted by $s_p \models \varphi$, iff, for every sequence $\gamma = \upsilon_1\upsilon_2 \dots \in (2^{V \setminus O})^\omega$ of valuations of the variables outside the control of p, the computation $comp(s_p, \gamma)$ resulting from s_p satisfies φ. We generalize the notion of winning from strategies to implementations (and, analogously, the notions of dominance and bounded dominance later in the paper), by defining that an implementation S is winning for φ iff the parallel composition of the strategies in S is winning (for their combined sets of inputs and outputs).

Synthesis. A property φ is *realizable* in an architecture A iff there exists an implementation that is winning for φ. We denote realizability by $A \boxdot\!\!\Rightarrow \varphi$.

Theorem 1. *[12] The question whether a property given by an LTL formula is realizable in an architecture with a single system process is 2EXPTIME-complete.*

Theorem 2. *[14] The question whether a property given by an LTL formula is realizable in an architecture is undecidable for architectures with two or more system processes.*

3 Preliminaries: Automata over Infinite Words and Trees

We assume familiarity with automata over infinite words and trees. In the following, we only give a quick summary of the standard terminology, the reader is referred to [9] for a full exposition.

A (full) *tree* is given as the set Υ^* of all finite words over a given set of directions Υ. For given finite sets Σ and Υ, a *Σ-labeled Υ-tree* is a pair $\langle \Upsilon^*, l \rangle$ with a labeling function $l : \Upsilon^* \to \Sigma$ that maps every node of Υ^* to a letter of Σ.

An *alternating tree automaton* $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ runs on Σ-labeled Υ-trees. Q is a finite set of states, $q_0 \in Q$ a designated initial state, δ a transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$, where $\mathbb{B}^+(Q \times \Upsilon)$ denotes the positive Boolean combinations

of $Q \times \Upsilon$, and α is an acceptance condition. Intuitively, disjunctions in the transition function represent nondeterministic choice; conjunctions start an additional branch in the run tree of the automaton, corresponding to an additional check that must be passed by the input tree. A run tree on a given Σ-labeled Υ-tree $\langle \Upsilon^*, l \rangle$ is a $Q \times \Upsilon^*$-labeled tree where the root is labeled with $(q_0, l(\varepsilon))$ and where for a node n with a label (q, x) and a set of children $child(n)$, the labels of these children have the following properties:

- for all $m \in child(n)$: the label of m is $(q_m, x \cdot \upsilon_m)$, $q_m \in Q, \upsilon_m \in \Upsilon$ such that (q_m, υ_m) is an atom of $\delta(q, l(x))$, and
- the set of atoms defined by the children of n satisfies $\delta(q, l(x))$.

A run tree is *accepting* if all its paths fulfill the acceptance condition. A *parity condition* is a function α from Q to a finite set of colors $C \subset \mathbb{N}$. A path is accepted if the highest color appearing infinitely often is even. The *safety condition* is the special case of the parity condition where all states are colored with 0. The *Büchi condition* is the special case of the parity condition where all states are colored with either 1 or 2, the *co-Büchi condition* is the special case of the parity condition where all states are colored with either 0 or 1. For Büchi and co-Büchi automata we usually state the coloring function in terms of a set F of states. For the Büchi condition, F contains all states with color 2 and is called the set of *accepting* states. For the co-Büchi condition, F contains all states with color 1 and is called the set of *rejecting* states. The Büchi condition is satisfied if some accepting state occurs infinitely often, the co-Büchi condition is satisfied if all rejecting states only occur finitely often. A Σ-labeled Υ-tree is *accepted* if it has an accepting run tree. The set of trees accepted by an alternating automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty iff its language is empty.

A *nondeterministic* automaton is an alternating automaton where the image of δ consists only of such formulas that, when rewritten in disjunctive normal form, contain at most one element of $Q \times \{\upsilon\}$ for every direction υ in every disjunct. A *universal* automaton is an alternating automaton where the image of δ contains no disjunctions. A *deterministic* automaton is an alternating automaton that is both universal and nondeterministic, i.e., the image of δ has no disjunctions and contains at most one element of $Q \times \{\upsilon\}$ for every direction υ.

A *word automaton* is the special case of a tree automaton where the set Υ of directions is singleton. For word automata, we omit the direction in the transition function.

4 Dominant Strategies

In game theory, strategic dominance refers to a situation where one strategy is better than any other strategy, no matter how the opponent plays. In the setting of reactive synthesis, *remorsefree dominance* [5] was introduced in order to accommodate situations that simply make it impossible to achieve the specified objective. For example, a module might have an input signal that resets its computation; if the reset signal is set too frequently it becomes impossible to complete the computation. In such a situation, we would expect the module to try to finish the computation as quickly as possible, to have the best chance to complete the computation before the next reset, but would

forgive the module for not completing the computation if the resets have made it impossible to do so.

Dominance can be seen as a weaker version of winning. A strategy $t : (2^I)^* \to 2^O$ *is dominated by* a strategy $s : (2^I)^* \to 2^O$, denoted by $t \preceq s$, iff, for every sequence $\gamma \in (2^{V \setminus O})^\omega$ for which the computation $comp(t, \gamma)$ resulting from t satisfies φ, the computation $comp(s, \gamma)$ resulting from s also satisfies φ. A strategy s is *dominant* iff, for all strategies t, $t \preceq s$. Analogously to the definition of winning implementations, we say that an implementation S is dominant iff the parallel composition of the strategies in S is dominant.

Finally, we say that a property φ is *admissible* in an architecture A, denoted by $A \boxdot\!\!\Rightarrow \varphi$, iff there is a dominant implementation.

Informally, a specification is admissible if the question whether it can be satisfied does not depend on variables that are not visible to the process or on *future inputs*. For example, the specification $\varphi = (\bigcirc a) \leftrightarrow b$, where a is an input variable and b is an output variable is not admissible, because in order to know whether it is best to set b in the first step, one needs to know the value of a in the second step. No matter whether the strategy sets b or not, there is an input sequence that causes *remorse*, because φ is violated for the chosen strategy while it would have been satisfied for the same sequence of inputs if the other strategy had been chosen.

Consider an architecture with a single process p. For a property given as an LTL formula, one can construct a nondeterministic parity tree automaton with an exponential number of colors and a doubly-exponential number of states in the length of the formula, such that the trees accepted by the automaton define exactly the dominant strategies. This can be done, following the ideas of [5], by first constructing a universal co-Büchi word automaton $\mathcal{A}_1$ that accepts a sequence in $(2^V)^\omega$ iff it satisfies the specification φ. The size of $\mathcal{A}_1$ is exponential in the length of φ. This automaton will be used to recognize situations in which the strategy satisfies the specification. Then, we construct a universal co-Büchi word automaton $\mathcal{A}_2$ that accepts a sequence in $(2^{V \setminus out(p)})^\omega$ iff it does *not* satisfy the specification φ for *any* choice of the outputs in $out(p)$. The size of $\mathcal{A}_2$ is also exponential in the length of φ. This automaton will be used to recognize situations in which the strategy does not need to satisfy the specification because no other strategy would either. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are combined in a product construction to obtain the universal co-Büchi word automaton $\mathcal{A}_3$, which accepts all sequences in $(2^V)^\omega$ that either satisfy φ or have the property that φ would be violated for all possible choices of the outputs $out(p)$. The size of $\mathcal{A}_3$ is still exponential in the length of φ. We then build a universal co-Büchi tree automaton $\mathcal{B}_1$ of the same size as $\mathcal{A}_3$ that accepts a $2^{out(p)}$-labeled $2^{inp(p)}$-tree iff the sequence along every branch and for every choice of the values of the variables in $V \setminus V(p)$ is accepted by $\mathcal{A}_3$. Converting $\mathcal{B}_1$ into an equivalent nondeterministic tree automaton $\mathcal{B}_2$ results in the desired nondeterministic parity tree automaton with an exponential number of colors and a doubly-exponential number of states in the length of the formula.

The synthesis of a dominant strategy thus reduces to checking tree automata emptiness and extracting a representation of some accepted tree as a finite-state machine. This can be done in exponential time in the number of colors and in polynomial time in the number of states [11]. For a matching lower bound, note that standard LTL syn-

thesis is already 2EXPTIME-hard [12]. Since every winning strategy is also dominant, we can reduce the standard synthesis problem to the synthesis of dominant strategies, by first checking the existing of a dominant strategy; if the answer is no, then no winning strategy exists. If the answer is yes, we synthesize a dominant strategy and verify (which can be done in polynomial time) whether it is winning. If it is winning, we have obtained a winning strategy, if not, then no winning strategy exists, because, otherwise, the synthesized strategy would not dominate the winning strategy, and, hence, would not be dominant.

Theorem 3. *The problem of deciding whether a property given as an LTL formula is admissible in a single-process architecture is 2EXPTIME-complete. A dominant strategy can be computed in doubly-exponential time.*

If the property is given as a deterministic automaton instead of as an LTL formula, admissibility checking only takes exponential time, because the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same size as the property automaton.

5 Synthesis of Environment Assumptions

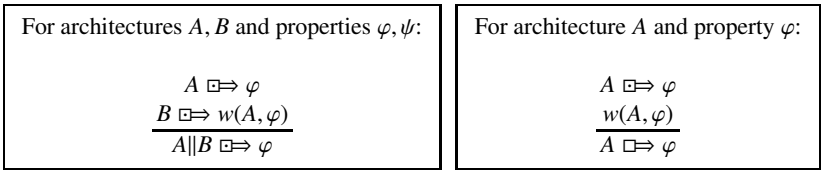
Standard compositional approaches for synthesis (cf. [7]) require the user to explicitly state the assumptions placed by the individual components on their environment. These assumptions need to be sufficiently strong so that each process can then be synthesized in isolation, relying only on the assumptions instead of the actual (and yet to be synthesized) implementation of the environment.

For admissible specifications, we can automatically construct the environment assumption. Since the dominant strategy defines the greatest set of environment behaviors for which the specification can be satisfied, the environment assumption is unique, and can in fact be represented by an automaton.

Theorem 4. *For an architecture A and a property φ such that $A \boxdot\!\!\Rightarrow \varphi$, there exists a unique weakest environment assumption, i.e., a unique largest set of sequences $w(A,\varphi) \subseteq (2^{V \setminus O})^\omega$ where $O = \bigcup_{p \in P} out(p)$, such that $A \square\!\!\Rightarrow w(A,\varphi) \rightarrow \varphi$. If φ is given as a deterministic parity word automaton, then there is a deterministic parity word automaton for $w(A,\varphi)$ with an exponential number of states. If φ is given as an LTL formula, the number of states is doubly-exponential in the length of the formula.*

Proof. We construct the deterministic parity automaton $\mathcal{A}_{w(A,\varphi)}$ for the weakest environment assumption as follows. Applying Theorem 3, we compute a dominant strategy s, represented as a transducer $\mathcal{A}_s = (Q_s, q_{s,0}, \delta_s : Q \times 2^{inp(p)} \rightarrow Q, \gamma_s : Q \rightarrow 2^{out(p)})$. Assume φ is given as a deterministic parity automaton $\mathcal{A}_\varphi = (Q_\varphi, q_{\varphi,0}, \delta_\varphi : Q \times 2^V \rightarrow Q, c)$. We combine $\mathcal{A}_s$ and $\mathcal{A}_\varphi$ to obtain the deterministic parity automaton $\mathcal{A}_\psi = (Q', q'_0, \delta', c')$ which recognizes all sequences that satisfy φ whenever the outputs of the process are chosen according to A_s.

- $Q' = (Q_s \times Q_\varphi) \cup \{\bot\}$,
- $q'_0 = (q_{s,0}, q'_{\varphi,0})$,



(a) Rule Inc-Synt (b) Rule A2R

Fig. 1. Rules Inc-Synt and A2R, implementing the incremental synthesis style.

- $\delta'((q_s, q_\varphi), i) = (q'_s, q'_\varphi)$ where $q'_s \in \delta_s(q_s, i \cap inp(p)), q'_\varphi \in \delta_\varphi(q_\varphi, i)\}$ if $i \cap out(q) = \gamma(q'_s)$, and $\delta'((q_s, q_\varphi), i) = \bot$, $\delta(\bot, i) = \bot$, otherwise.
- $c'(q_s, q_\varphi) = c(q_\varphi), c'(\bot) = 0$.

The language of $\mathcal{A}_\psi$ is the unique weakest environment assumption: suppose that there exists an environment assumption ψ' with $\mathcal{L}(\mathcal{A}_\psi) \subsetneq \psi'$, then there is a sequence γ in $\psi' \setminus \mathcal{L}(\mathcal{A}_\psi)$ for which there exists a strategy t such that the computation resulting from γ and t satisfies φ, while the computation resulting from γ and s does not satisfy φ. This contradicts that s is dominant.

□

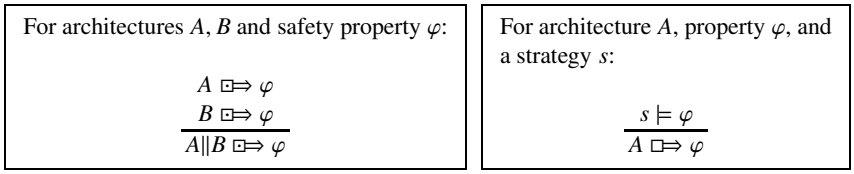
Theorem 4 can be used to synthesize a distributed system *incrementally*, i.e., by constructing one process at a time and propagating the environment assumptions. This synthesis style corresponds to the repeated application of Rule Inc-Synt, shown in Figure 1a: in order to prove the admissibility of a specification φ in an architecture $A \| B$, we show that φ is admissible in A, and the resulting environment assumption is admissible in B. Once the full system has been synthesized, we verify that the remaining environment assumption is *true*, which proves that the specification holds for all possible inputs. This last step corresponds to an application of Rule A2R, shown in Figure 1b.

Theorem 5. *Rules* Inc-Synt *and* A2R *are sound.*

6 Compositional Synthesis for Safety Properties

With the incremental synthesis approach of Rules Inc-Synt and A2R, we reduce the synthesis problem for the distributed system to a sequence of admissibility checks over individual processes. The disadvantage of incremental synthesis is its inherent sequentiality: we cannot consider processes in parallel; additionally, each application of Rule Inc-Synt increases the size of the specification.

In this section, we introduce a *compositional* approach, where the processes are considered *independently* of each other. Figure 2a shows the compositional synthesis rule Safety-Comp-Synt. In order to synthesize an implementation for specification φ in the distributed architecture $A_1 \| A_2$, we check whether φ is admissible on both A_1 and A_2. If φ is admissible on both A_1 and A_2, it is also admissible on $A_1 \| A_2$. For the final check whether the specification is satisfied for all environment behaviors, we model check the



(a) Rule Safety-Comp-Synt (b) Rule MC

Fig. 2. Rules Safety-Comp-Synt and MC, implementing the compositional synthesis style.

resulting dominant strategy. This last step corresponds to an application of Rule MC, shown in Figure 2b.

Note that Rule Safety-Comp-Synt is restricted to safety properties. The rule is in fact not sound for liveness properties. Consider $\varphi = ((\Diamond a) \leftrightarrow (\Box\Diamond c)) \wedge ((\Diamond b) \leftrightarrow (\Box\Diamond c))$, where a is the output of A_1, b is the output of A_2, and c is the output of the external environment of $A_1 \| A_2$. A dominant strategy s_1 for A_1 is to wait for the first b and then, in the next step, output a. Suppose there are, on some input sequence, infinitely many c and some b, or only finitely many c, then s_1 satisfies φ. On the other hand, if there are infinitely many c but no b, then φ is violated no matter what strategy A_1 chooses. Hence, s_1 is dominant. Likewise, a dominant strategy for A_2 is to wait for the first a and then, in the next step, produce a b. However, $A_1 \| A_2$ does not have a dominant strategy for φ, because we require $A_1 \| A_2$ to predict whether or not the environment will set c to *true* infinitely often. Any strategy will fail this objective on at least some input sequence; however, given such an input sequence there is always a strategy that makes the correct prediction for that particular sequence.

In the following, we prove that Rule Safety-Comp-Synt is sound for safety properties. We will adapt Rule Safety-Comp-Synt to arbitrary properties in Section 7. The reason for the soundness of Rule Safety-Comp-Synt is that the parallel composition of two dominant strategies is again dominant.

Lemma 1. *For a safety property φ it holds that if s_1 is dominant for A_1 and s_2 is dominant for A_2, then $s_1 \| s_2$ is dominant for $A_1 \| A_2$.*

Proof. Let O_1, O_2, and O_{12} be the output variables of the processes in A_1, A_2, and A_{12}, respectively, and let V be the set of variables in all three architectures. Suppose, by way of contradiction, that there exists a sequence $\gamma \subseteq (2^{V \setminus O_{12}})^\omega$ of valuations of variables outside the control of the processes in $A_1 \| A_2$ such that the computation $\sigma = \mathit{comp}(s_1 \| s_2, \gamma)$ resulting from $s_1 \| s_2$ does not satisfy φ, but there exists a strategy t such that the resulting computation $\sigma' = \mathit{comp}(t, \gamma)$ satisfies φ. We pick the smallest prefix $\delta \cdot \eta$ of σ, where $\delta \in (2^V)^*, \eta \in 2^V$ such that every infinite extension of $\delta \cdot \eta$ violates φ but there is an infinite extension σ'' of δ that agrees with σ on the variables $V \setminus O_{12}$ outside the control of the processes in $A_1 \| A_2$ and that satisfies φ. Such a prefix exists because φ is a safety property. The prefix cannot be the empty sequence, because otherwise all sequences that agree with σ on $V \setminus O_{12}$, including σ', would violate φ. The last position η of the prefix contains decisions of both s_1 and s_2. We make the following case distinction:

– There is an infinite extension σ''' of $\delta{\cdot}\eta'$ for some η' with $\eta' \cap (V \setminus O_1) = \eta \cap (V \setminus O_1)$ such that $\sigma''' \models \varphi$, i.e., the violation of φ is the fault of strategy s_1. In this case, s_1 is not dominant, because the sequence that results from restricting σ''' to the variables $V \setminus O_1$ outside the control of A_1 causes s_1 to violate φ, while an alternative strategy, producing the outputs of σ''', would satisfy φ.
– There is no infinite extension σ''' of $\delta{\cdot}\eta'$ for some η' with $\eta' \cap (V \setminus O_1) = \eta \cap (V \setminus O_1)$ such that $\sigma''' \models \varphi$, i.e., the violation of φ is (at least also) the fault of strategy s_2. In this case, s_2 is not dominant, because the sequence that results from restricting σ'' to the variables $V \setminus O_2$ outside the control of A_2 causes causes s_2 to violate φ, while an alternative strategy, producing the outputs of σ'', would satisfy φ.

Either case contradicts the assumption that s_1 and s_2 are dominant. □

In light of the observation that Rule Safety-Comp-Synt cannot be generalized to liveness properties, it is not surprising that Lemma 1 does not hold for liveness properties either. Consider the specification $(\Diamond a) \wedge (\Diamond b)$, where a is the output of A_1 and b is the output of A_2. A dominant strategy s_1 for A_1 is to wait for the first b and then, in the next step, output a. The strategy guarantees the specification on all paths that have a b somewhere; no strategy for A_1 satisfies the specification on paths without a b. Likewise, a dominant strategy for A_2 is to wait for the first a and then, in the next step, produce a b. The composition $s_1 \| s_2$, will, however, never output an a or b and therefore violate the specification, despite the fact that even winning strategies exist, such as the strategy that immediately outputs a and b.

Lemma 1 implies the soundness of Rule Safety-Comp-Synt. The soundness of Rule MC is trivial, as the strategy s is guaranteed to satisfy the specification φ.

Theorem 6. *Rules* Safety-Comp-Synt *and* MC *are sound.*

7 Compositional Synthesis for Liveness Properties

We saw in the preceding section that the soundness of Rule COMP-SYNT breaks for liveness properties, because the composition of two dominant strategies is not necessarily also dominant. In this section, we propose an alternative notion of admissibility, which we call *bounded* admissibility, which is preserved under composition.

We motivate bounded dominance with the example from Section 6. Consider again the property $\varphi = (\Diamond a) \wedge (\Diamond b)$ where a is the output of A_1 and b is the output of A_2. We introduced the dominant strategy s_1 for A_1, which waits for the first b before outputting a. Strategy s_1 is problematic, because it is dominant for A_1, but does not result in a dominant strategy $s_1 \| s_2$ for $A_1 \| A_2$, when combined with the corresponding strategy s_2 for A_2, which waits for the first a before outputting b.

The problem is that both s_1 and s_2 postpone their respective output *indefinitely*, because they both wait for the other strategy to start. Bounded dominance refines the valuation of the strategy by counting the number of steps it takes before a and b become true. This number is compared to a fixed bound n, say $n = 5$. Strategy s_1 is *not* dominant with respect to bound n, because it may unnecessarily exceed the bound. There is an n-dominant strategy s_1', which sets a in the very first step and therefore meets the bound

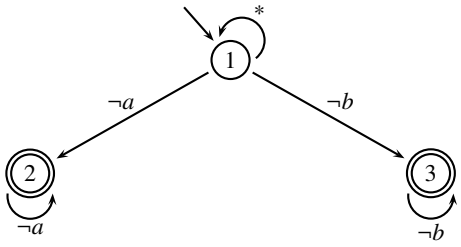


Fig. 3. Universal co-Büchi automaton for the LTL formula $\varphi = \Box((\Diamond a) \wedge (\Diamond b))$. The states depicted with double circles (2 and 3) are the rejecting states in F.

whenever possible, i.e., as long as b arrives within 5 steps. The corresponding strategy s_2' for A_2, which outputs b in the first step, is n-dominant for A_2. Replacing s_1 and s_2 with s_1' and s_2' solves the problem: The combined strategy $s_1 \| s_2$ is n-dominant for $A_1 \| A_2$.

We prepare the definition of bounded dominance by defining the *measure* of a computation. The measure captures how quickly a strategy makes progress with respect to a liveness property. We define the measure with respect to a representation of the specification as a universal co-Büchi automaton. Such an automaton can be produced with standard LTL-to-Büchi translation algorithms, by first constructing a nondeterministic Büchi automaton for the negation of the specification and then dualizing the automaton to obtain a universal co-Büchi automaton for the complement language [13, 8]. If the specification is a conjunction of properties, the size of the automaton is linear in the number of conjuncts: we apply the translation to the individual conjuncts, resulting in automata with an exponential number of states in the length of the conjunct, and then compose the automata by branching (universally) from the initial state into the otherwise disjoint subautomata for the conjuncts.

Lemma 2. *Let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ be an LTL formula that consists of a conjunction of properties. There is a universal co-Büchi automaton that accepts exactly the computations that satisfy φ, such that the automaton consists of subautomata for the individual conjuncts that only overlap in the initial state. The size of the automaton is exponential in the length of the largest conjunct and linear in the number of conjuncts.*

The automaton accepts a computation iff the number of visits to rejecting states is finite on every path of the run tree. We define the *measure* of the computation σ, denoted by $measure_\varphi(\sigma)$ as the supremum of the number of visits to rejecting states over all paths of the run tree of the automaton for φ. If there is no run tree, we set the measure to ∞.

As an example, consider $\varphi = \Box((\Diamond a) \wedge (\Diamond b))$. The universal co-Büchi automaton for φ is shown in Figure 3. The computation $\{a, b\}^\omega$ has measure 0, because the run tree only has a single path, labeled everywhere with state 1. The computation $\emptyset\{a\}\{a, b\}^\omega$ has measure 2: There are three paths, an infinite path labeled with state 1 everywhere, and two finite paths, one labeled with state 1 followed by state 2, and one labeled with state

1, followed by two times state 3. The number of visits to rejecting states are thus 0, 1, and 2, respectively, and the supremum is 2.

Let n be a fixed natural number. We say that a strategy $t : (2^I)^* \to 2^O$ *is dominated with bound n* (or short: *n-dominated*) by a strategy $s : (2^I)^* \to 2^O$, denoted by $t \trianglelefteq_n s$, iff, for every sequence $\gamma \in (2^{V \setminus O})^\omega$ for which the measure of the computation $comp(t, \gamma)$ resulting from t is less than or equal to n, the measure of the computation $comp(s, \gamma)$ resulting from s is also less than or equal to n. A strategy s is *n-dominant* iff, for all strategies t, $t \trianglelefteq_n s$. A property φ is *n-admissible* in an architecture A, denoted by $A \diamond\!\!\Rightarrow_n \varphi$, iff there is an n-dominant implementation.

If the universal automaton is a *safety* automaton, then dominance and n-dominance are equivalent. Since the safety automaton does not have any rejecting states, the measure is either 0, if the property is satisfied, or ∞, if the property is violated and there is, therefore, no run tree. Hence, the definitions of dominance and bounded dominance agree for any choice of the bound.

As an example property that has a dominant strategy but no n-dominant strategy for any bound n, consider $(\Diamond a) \leftrightarrow (\Diamond b)$, where a is the input and b the output. This property can be satisfied for every possible input by waiting for an a before setting the b. For example, setting b in the step after the first a is observed is a winning and therefore dominant strategy. However, this strategy, as well as any other strategy that waits for an a before setting b, is not n-dominant for any choice of n: consider the situation where a occurs exactly every n steps; then the measure of the strategy would be $n + 1$, while an alternative strategy that produces a b every n steps has only measure n.

Note that bounded admissibility does not imply admissibility; any specification of the form $(\Diamond a) \wedge (\neg a) \wedge (\bigcirc \neg a) \wedge \varphi$, where a is an output, is 1-admissible, because it is impossible to achieve a measure ≤ 1; obviously, there are formulas φ for which this specification is not admissible.

Bounded dominance can be checked with a small variation of the construction from Section 4: we simply modify the universal automaton $\mathcal{A}_1$, which verifies that strategy s_p achieves its goal, as well as the universal automaton $\mathcal{A}_2$, which checks whether any alternative strategy would achieve the goal, by counting the number of visits to rejecting states up to n.

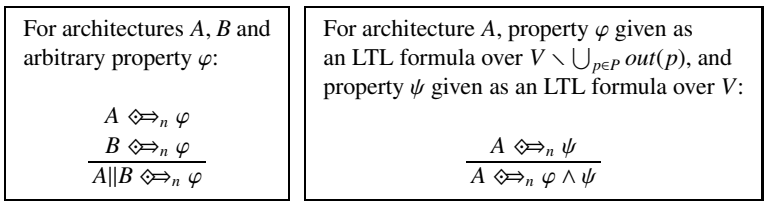
Theorem 7. *For a fixed bound n, the problem of deciding whether a property given as an LTL formula is n-admissible in a single-process architecture is 2EXPTIME-complete. An n-dominant strategy can be computed in doubly-exponential time.*

Rule General-Comp-Synt, shown in Figure 4a, generalizes the compositional synthesis approach from Rule Safety-Comp-Synt to general properties. Because Rule General-Comp-Synt is based on bounded admissibility $\diamond\!\!\Rightarrow_n$ instead of standard admissibility $\boxdot\!\!\Rightarrow$, Lemma 1 now holds for general properties:

Lemma 3. *For an arbitrary property φ it holds that if s_1 is n-dominant for A_1 and s_2 is n-dominant for A_2, then $s_1 \| s_2$ is n-dominant for $A_1 \| A_2$.*

The proof of Lemma 3 is analogous to the proof of Lemma 1. Lemma 3 implies the soundness of Rule General-Comp-Synt.

Theorem 8. *Rule* General-Comp-Synt *is sound.*



(a) Rule General-Comp-Synt (b) Rule Decomp

Fig. 4. Rules General-Comp-Synt and Decomp.

8 Property Decomposition

Specifications are usually given as a conjunction of properties. The goal of *property decomposition* is to avoid analyzing all properties in the synthesis of every process, and instead only focus on a small set of "relevant" properties for each process.

In general, it is not sound to leave out conjuncts when checking the admissibility of the specification for some process, even if, overall, every conjunct is "covered" by some process. The problem is that the missing conjuncts may invalidate admissibility. Consider, for example, the properties $\varphi = \square(a \leftrightarrow \bigcirc b)$ and $\psi = \square(c \leftrightarrow \bigcirc b)$, where a is an input variable, and b and c are output variables. Individually, both φ and ψ are admissible, but their conjunction $\varphi \wedge \psi$ is not: in order to set the value of c correctly, a dominant strategy would need to predict the future input a.

Conjuncts that do not refer to output variables enjoy, however, the following monotonicity property: if φ does not refer to the output variables, then for every (n-)admissible property ψ it holds that $\varphi \wedge \psi$ is also (n-)admissible.

Theorem 9. *Let φ be an LTL formula over $V \setminus \bigcup_{p \in P} out(p)$, and ψ an LTL formula over V. Then it holds that if ψ is (n-)admissible, then $\varphi \wedge \psi$ is also (n-)admissible.*

Proof. Suppose, by way of contradiction, that there is a strategy $s : (2^I)^* \to 2^O$ that is dominant for ψ, but not for $\varphi \wedge \psi$. Then there exists a strategy t and a sequence $\gamma \in (2^{V \setminus O})^\omega$ of variable valuations that are not under the control of the process, such that the computation resulting from t satisfies $\varphi \wedge \psi$ and the computation resulting from s does not. Since φ only refers to uncontrollable variables, the truth value of φ is determined by γ; we therefore know that φ must also be satisfied by the computation resulting from s. Hence, ψ must be violated on the computation resulting from s, while it is satisfied by the computation resulting from t. This contradicts the assumption that s is dominant for ψ.

For bounded admissibility assume, analogously, that there is a strategy s that is n-dominant for ψ, but not for $\varphi \wedge \psi$. Then there exists a strategy t and a sequence $\gamma \in (2^{V \setminus O})^\omega$ such that $measure_{\varphi \wedge \psi}(comp(t, \gamma)) \leq n < measure_{\varphi \wedge \psi}(comp(s, \gamma))$. Since the subautomata for the conjuncts only intersect in the initial state, every path of the run tree is, starting with the second state, either completely in the subautomaton for φ or in the subautomaton for ψ. Since φ only refers to uncontrollable variables, the paths, and, hence, the number of visits to rejecting states in the subautomaton of φ are the same

for $comp(s, \gamma)$ as for $comp(t, \gamma)$. Hence, there must be some path in the subautomaton for ψ where $comp(s, \gamma)$ visits rejecting states more than n times, while $comp(t, \gamma)$ visits rejecting states less than or equal to n times. This contradicts the assumption that s is n-dominant for ψ. □

Theorem 9 can be used to eliminate conjuncts that do not refer to output variables. This decompositional synthesis style corresponds to applications of Rule Decomp, shown in Figure 4b.

9 The Compositional Synthesis Algorithm

Putting the results from the preceding sections together, we obtain the following synthesis algorithm. For an architecture $A = A_1 || A_2 || \ldots$ composed of multiple single-process architectures and a specification φ, given as a conjunction $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \varphi_m$ of LTL formulas, we do the following:

1. Applying Rule General-Comp-Synt, check for all subarchitectures A_i whether $A_i \diamond\!\!\Rightarrow_n \varphi$; if so, synthesize a dominant (or n-dominant, for liveness properties) strategy.
   - for this purpose, use Rule Decomp to identify a subset $C \subseteq \{1, 2, \ldots, m\}$ of the conjuncts such that $A_i \diamond\!\!\Rightarrow_n \bigwedge_{j \in C} \varphi_j$, and
   - compose the n-dominant strategies according to Lemma 3.
2. Apply Rule MC to check whether the resulting strategy satisfies φ. If yes, a correct implementation has been found.

For specifications given as LTL formulas, the complexity of the compositional synthesis algorithm is doubly-exponential in the length of the formula. Since the synthesis of the strategies for the subarchitectures is independent of each other, the complexity of finding the strategies is linear in the number of processes; the complexity of composing the strategies and checking the resulting strategy is exponential in the number of processes.

10 Conclusions

We have presented an approach for the synthesis of distributed systems from temporal specifications. For admissible specifications, the complexity of our construction is dramatically lower than that of previously known algorithms. Since the synthesis method is compositional, it can easily be parallelized. The constructed implementations are modular and much smaller than those constructed by previous approaches that work on a "flattened" state space. The construction is furthermore universally applicable to all system architectures, including the large class of architectures for which the standard synthesis problem is undecidable.

References

1. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proc. DATE. pp. 1188–1193 (2007)
2. Bloem, R.P., Gamauf, H.J., Hofferek, G., Könighofer, B., Könighofer, R.: Synthesizing robust systems with RATSY. In: Association, O.P. (ed.) SYNT 2012. vol. 84, pp. 47 – 53. Electronic Proceedings in Theoretical Computer Science (2012)
3. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV. LNCS, vol. 7358, pp. 652–657. Springer (2012)
4. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Intl. Congr. Math. pp. 23–25. Upsala (1963)
5. Damm, W., Finkbeiner, B.: Does it pay to extend the perimeter of a world model? In: Butler, M., Schulte, W. (eds.) FM. Lecture Notes in Computer Science, vol. 6664, pp. 12–26. Springer (2011)
6. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS. Lecture Notes in Computer Science, vol. 6605, pp. 272–275. Springer (2011)
7. Finkbeiner, B., Schewe, S.: Semi-automatic distributed synthesis. In: ATVA 2005. pp. 263–277. Springer Verlag (2005)
8. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer 15(5-6), 519–539 (2013)
9. Grädel, E., Thomas, W., Wilke, Th. (eds.): Automata, Logics, and Infinite Games, LNCS, vol. 2500. Springer-Verlag (2002)
10. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019 (1995)
11. Jurdziński, M.: Small progress measures for solving parity games. In: Proc. STACS. pp. 290–301 (2000)
12. Kupferman, O., Vardi, M.Y.: Synthesis with incomplete informatio. In: Proc. of ICTL (1997)
13. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October, Pittsburgh, PA, USA. pp. 531–540 (2005)
14. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. FOCS'90. pp. 746–757 (1990)
15. de Roever, W.P., Langmaack, H., Pnueli, A. (eds.): Compositionality: The Significant Difference, International Symposium, COMPOS'97, Lecture Notes in Computer Science, vol. 1536. Springer (1998)