What You Really Need To Know About Your Neighbor*

Werner Damm
CvO Universität Oldenburg,
26111 Oldenburg
werner.damm@offis.de

Bernd Finkbeiner
Universität des Saarlandes,
Campus E1 3, 66123 Saarbrücken
finkbeiner@cs.uni-saarland.de

Astrid Rakow
CvO Universität Oldenburg,
26111 Oldenburg
a.rakow@uni-oldenburg.de

A fundamental question in system design is to decide how much of the design of one component must be known in order to successfully design another component of the system. We study this question in the setting of reactive synthesis, where one constructs a system implementation from a specification given in temporal logic. In previous work, we have shown that the system can be constructed compositionally, one component at a time, if the specification admits a dominant strategy for each component. In this paper, we generalize the approach to settings where dominant strategies only exist under certain assumptions about the future behavior of the other components. We present an incremental synthesis method based on the automatic construction of such assumptions.

1 Introduction

We investigate the following fundamental research question in system design: Consider a system architecture A composed of two processes p_h, p_l of different priorities high and low, all with defined interfaces $inp(A)$, $out(A)$, $inp(p_h)$, $out(p_h)$, $inp(p_l)$, $out(p_l)$ and objectives φ_A, φ_{p_h}, φ_{p_l} expressed in linear time temporal logic using the interfaces of A, p_h, p_l. How much does p_h need to know about p_l so that

(1) both jointly realize φ_A whenever possible,

(2) p_h can realize its objective whenever possible,

(3) p_l only sacrifices achieving its objectives when this is the only way to achieve (1) and (2)?

We provide a rigorous formulation of this research question, and give algorithms to compute the insight p_h must have into p_l. We then show how this can be used to generate best-in-class strategies for p_h and p_l, in the sense that if there is a strategy at all for p_h which in the context of p_l achieves both (1) and (2), then so will the best-in-class strategy for p_h, and if there is a strategy at all for p_l which achieves (3), then so will the best-in-class strategy for p_l. The algorithm works for any number of processes with priorities henceforth represented by a partial order $<$.

Such system design questions arise naturally in many application areas, with A being a system of systems, φ_A representing overarching control objectives, and constituent systems $p_1, \ldots, p_n$ with own local control objectives, where $<$ reflects the degree of criticality of the subsystem for the overall system. In such systems, the notion of neighborhood has its classical interpretation based on physical proximity. One example of such systems are smart grids, where subsystems represent different types of energy providers and consumers, and the highest priority subsystem is responsible for maintaining stability of the subgrid in spite of large fluctuations of inflow and outflow of energy. The running example we use comes from cooperative autonomous vehicle control, where the overall objectives relate to safety,

*This work has been partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

Submitted to:
SYNT 2016

avoiding congestion, fuel and CO_2 reduction etc, and local objectives relate to reaching destinations in given time frames, reducing fuel consumption, and executing certain maneuvers. We focus here on simple objectives such as completing a maneuver with cooperation from neighboring cars.

As pointed out in [10, 11], for systems of systems controlling physical systems as in the two examples above, there will be no winning strategies, because rare physical events such as stemming from extreme weather conditions or physical system failures can occur, making it impossible to even achieve overarching safety objectives. No electronic stability system of a car would be able to still guarantee safety in unexpected icy road conditions in a dynamical situation with already stretched system limits. We introduced the notion of *(remorse-free) dominant strategies* to compare strategies in such applications where a winning strategy can not be found, making precise what has been called "best-in-class" strategies above. Such a strategy will – for every sequence of environment actions – be at least as good in achieving the system objectives as any other strategy. We can now make precise the addressed research question: how much insight does p_h need into the plans of p_l, so that there is a dominant strategy for p_h to achieve (1) and (2) above?

In general, local strategies must cater for needs of neighboring systems, so as to enable synthesis of (remorse-free) dominant systems. Consider e.g. a two car system on the highway, with the ego car wanting to reach the next exit, and having a neighboring car on the rightmost lane. There is no remorse-free dominant strategy of the ego car to reach the exit, unless the car to its right cooperates: if the ego car accelerates to overtake the car, the car on right might do the same; similarly, an attempt to reach the exit by slowing down might be blocked by the car to the right. Thus ego needs insight into the future moves of the other car. In fact, if the other car would signal, that its next move will be to decelerate, then the ego car can bet on accelerating so as to overtake the car and reach the exit. If, on the other hand, the other car would promise to accelerate, then ego could decelerate and thus change lane in order to reach the exit. The weakest assumption for ego to have a dominant strategy will not only consider the next move of the other car, but give the other car as much flexibility as possible, as long as a lane change will still be possible prior to reaching the exit. We formalize these as what we call *assumption trees* of ego, which branches at each point in time into the set of possible futures of settings of the speed control of the other car s.t. the ego car has a dominant strategy in all such futures, and show that we can compute the most general assumption tree under which ego has a dominant strategy. A strategy for the other car is then not only determining the speed control of other, but also picking only evolutions which are compatible with ego's assumption tree. Rather than communicating the choice of such future moves through an extended interface, we provide ego with a copy of the strategy of other, so that ego can determine the selected future by simulating the strategy of the other car.

In general, such assumption trees can have unbounded depth. Consider the slight variant of the above example, where ego only wants to change lanes (no exit targeted) and has as secondary objective to always keep its speed. The most general assumption tree of ego will then, at each depth, have choices of futures where the other car will always keep its speed, forcing ego to either accelerate or decelerate to at least meet its higher priority objective of changing lane, and those futures where eventually other either breaks or accelerates, in which case ego can achieve both its objectives.

In general, we assume to have n processes ordered by criticality $<$. Intuitively, if p_h is more critical than p_l ($p_h > p_l$), then we would want p_l to adapt its behavior so as to be compliant to the assumption tree of p_h. For a totally ordered set of processes $p_1 > p_2 > \ldots > p_n$ we thus generate assumption trees inductively starting with p_1, and then propagate this along the total order. Assuming the overall environment of $A = \{p_1, \ldots, p_n\}$ meets the assumption tree of p_n, we can then synthesize for each p_j a remorse-free dominant strategy meeting the canonical generalization of (1),(2),(3) above to n processes. If processes are only partially ordered, we perform this algorithm for clusters of processes with same

priority.

Related Work We introduced the notion of remorse-free dominant strategies in [10], and provided methods for compositional synthesis of remorse-free dominant strategies in [11]. Compositional approaches have been studied in the setting of assume-guarantee synthesis [3], where the synthesized strategies are guaranteed to be robust with respect to changes in other components, as long as the other components do not violate their own, local, specifications. In contrast, in this paper, we derive assumptions to ensure cooperation. The automatic synthesis of strategies for reactive systems goes back to the seminal works by Church, Büchi, Landweber, and Rabin in the 1960s [5, 2, 24], and by Pnueli and Rosner in the 1980s [22]. For distributed systems, the synthesis problem is known to be undecidable in general [23], and decidable under certain assumptions such as well-connected system architectures [16]. The problem that specifications usually must be strengthened with environment assumptions to become realizable has been recognized by several authors. The construction by Chatterjee et al [4] directly operates on the game graph. A safety assumption is computed by removing a minimal set of environment edges from the graph. A liveness assumption is then computed by putting fairness assumptions on the remaining edges. Liu et al [18] mine counterexamples to realizability to find new assumptions. A key difference between the assumptions generated in these approaches and the assumptions of this paper is that we compute assumptions for the existence of dominant strategies, not for the existence of winning strategies. As a result, our assumptions are weaker. In particular, it is in general not necessary to restrict the behavioral traces of the other components, only their branching structure. Brenguier et al [1] introduce an approach called assume-admissible synthesis, based on a notion of admissible strategies that is closely related to our notion of dominant strategies. The difference is that in assume-admissible synthesis it is assumed a-priori that the other components also apply admissible strategies, while we make no such assumption.

To use formal methods and in particular formal synthesis methods for coordinated vehicle maneuvers has been proposed in among others [15, 14, 28, 12, 8, 13, 27, 21]. [15] searches for strategies controlling all vehicles, and employs heuristic methods from artificial intelligence such as tree-search to determine strategies for coordinated vehicle movements. An excellent survey for alternative methods for controlling all vehicles to perform collision free driving tasks is given in [14]. Both methods share the restriction of the analysis to a small number of vehicles, in contrast to our approach, which is based on safe abstractions guaranteeing collision freedom in achieving the driver's objectives taking into account the complete traffic situation. In [6], the authors show that LTL formulas are expressive enough to express typical traffic-flow optimization objectives, and use formal synthesis methods to automatically generate control commands for access control on highway systems. In [28], the authors describe a methodological approach for decomposing the synthesis problem, which, however, has already been described in a previous journal publication by the proposers [12]. It additionally presents a tool for constructing the finite abstraction; our previous work in [8, 13] goes beyond the capabilities of this tool in also allowing for non-linear dynamics. In [27], the authors provide an improved two-level approach for control synthesis that however suffers from a flaw in constructing the abstraction relation. In [21] the authors provide robust finite abstractions with bounded estimation errors for reducing the synthesis of winning strategies for LTL objectives to finite state synthesis and demonstrate the approach for an aerospace control application; however, this approach does not cover cooperative maneuvers. We finally mention that in the aerospace domain, high-level objectives such as maximizing throughput and energy efficiency while maintaining safety have led to new concepts for air-traffic control, such as free flight. There is a significant body of work on the verification of the safety of such control strategies as well as

on design rules that ensure safety [26, 19, 20, 25, 7, 9].

Outline Our paper is structured as follows. We introduce preliminaries on strategy synthesis and tree automata in Section 2 and 3. Section 5 shows how to compute assumptions for cooperation. Section 6 provides the incremental distributed synthesis procedure. As a running example to illustrate our notions, we use the *side by side* example, where a car A starts side by side with car B and A's goal is to eventually be before or after B. We conclude in Sect. 7.

2 Synthesis of Distributed Systems

We consider complex multi-component systems at an early design stage, where the system architecture and the design objectives are known, but the components have not been implemented yet. We are interested in synthesizing an implementation for a given system architecture A and a specification 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 $V(p) = inp(p) \cup out(p)$. Variables $V_I = V \setminus \bigcup_{p \in P} out(p)$ that are not the output of any system process are called *external inputs*. We assume that the external inputs are available to every process, i.e., $V_I \subseteq inp(p)$ for every $p \in P$.

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 $A_1 \parallel A_2 = (P_1 \cup P_2, V, p \mapsto \text{if } p \in P_1 \text{ then } inp_1(p) \text{ else } inp_2(p), p \mapsto \text{if } p \in P_1 \text{ then } out_1(p) \text{ else } out_2(p))$.

Example: We use as running example the scenario from the introduction, where a car is side-by-side to another car and has to reach an exit. To be usable as running example, we strip this down to the bare essentials.

Our formal model will consider this two car system as an architecture consisting of two "processes", which we call ego and other. We restrict ourselves to finite state systems, and abstract the physical notion of positions of the two cars on adjacent lanes of a highway into the states *side_by_side*, and *was_aside*, which are interpreted from the perspective of the ego car.

Each car can control its own dynamics by accelerating, keeping the current speed, or decelerating. These actuators are captured in our formal model as output variables. E.g. for the process ego its output variables *out*(ego) are a_e, k_e, d_e with the obvious meaning. We consider fully informed processes, hence ego has *out*(other) as input variable.

We capture the effect of the setting of output variables and input variables of a process on the system state in what we call world models. Figure 1 shows the world model of the ego car. Initially, both cars are assumed to be side by side. State transitions are labeled by pairs of the chosen output of the process and the observed input of the process. E.g. if ego chooses to keep its speed, and the other car keeps its speed, too, then the system remains in state side_by_side. If, on the other hand, ego chooses to accelerate, while other keeps its velocity, the system transitions to state was_aside. Multiple labels are just shorthand for multiple transitions with unique labels. Note that the world model of other is identical to that of ego

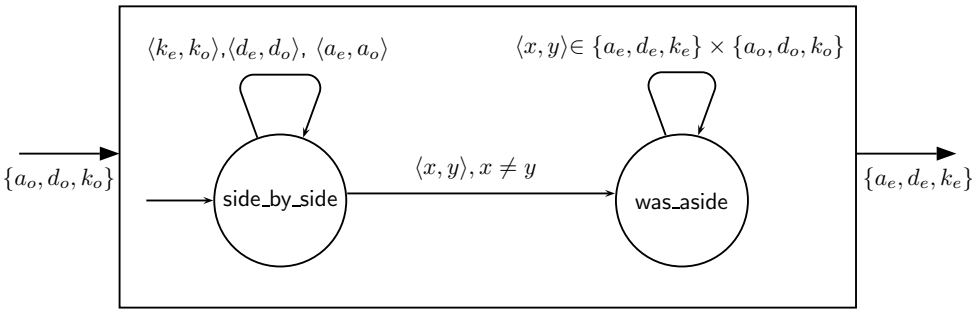


Figure 1: World model

except for exchanging inputs and outputs. From a control-theoretic perspective, the relative position of cars is a plant variable, which is influenced by the actuators of both cars. □

We now give the formal definition of the world model of a process p.

Let $\mathbf{S}$ be a set of (system-) states. A world model M for process p with input variables $inp(p)$ and output variables $out(p)$ is an input deterministic transition system over $\mathbf{S}$ with designated initial state and transitions labeled by pairs $\langle o, i\rangle$ where $i \in inp(p)$ and $o \in out(p)$. Formally $M(p) = (\mathbf{S}, E, L, s_0)$ with $E \subseteq \mathbf{S} \times \mathbf{S}$, $L : E \to out(p) \times inp(p)$, $s_0 \in \mathbf{S}$ s.t. $(s, s_1) \in E \wedge (s, s_2) \in E \wedge L(s, s_1) = L(s, s_2) \Rightarrow s_1 = s_2$

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 $\alpha_p : 2^I \to 2^{inp(p)}$ fills in for process p what q contributes to the p's input, that is $\alpha_p(\epsilon) = \epsilon$ and $\alpha_p(\upsilon_0\upsilon_1 \ldots \upsilon_k) = ((\upsilon_0 \cup s_q(\epsilon)) \cap inp(p))((\upsilon_1 \cup s_q(\alpha_q(\upsilon_0))) \cap inp(p)) \ldots ((\upsilon_k \cup s_q(\alpha_q(\upsilon_1\upsilon_2 \ldots \upsilon_{k-1}))) \cap inp(p))$, and, analogously, $\alpha_q(\epsilon) = \epsilon$ and $\alpha_q(\upsilon_0\upsilon_1 \ldots \upsilon_k) = ((\upsilon_0 \cup s_p(\epsilon)) \cap inp(q))((\upsilon_1 \cup s_p(\alpha_p(\upsilon_0))) \cap inp(q)) \ldots ((\upsilon_k \cup s_p(\alpha_p(\upsilon_1\upsilon_2 \ldots \upsilon_{k-1}))) \cap inp(q))$.

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

Specification We use linear-time temporal logic (LTL) to specify properties. In the following, we will denote the `next-time` operator with $\bigcirc$, `globally` with $\Box$ and `eventually` with $\Diamond$. For a computation σ and an ω-regular language φ, we write $\sigma \models \varphi$ if σ satisfies φ. Design objectives are often given as a *prioritized* conjunction of LTL formulas $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$, where φ_1 is the most important objective and φ_n is the least important objective. For a priority k, with $1 \leq k \leq n$, we consider the partial conjunction $\varphi^k = \bigwedge_{i=1\ldots k} \varphi_i$ and say that σ satisfies φ *up to priority k* if $\sigma \models \varphi^k$.

Example: In the side by side example, the control objective of the ego car is to reach a state where it can change lane, i.e., to eventually drive the plant into a state different from side_by_side. From the perspective of ego, the actuator settings of the other car are uncontrollable disturbances. Suppose that the ego car has the secondary objective to reduce fuel consumption, which very abstractedly can be captured

by avoiding accelerating and decelerating altogether, in other words to always output k_e. Suppose finally that other is owned by a driver who may avoid getting tired by changing speed every now and then. We capture these overall control objectives as list of prioritized LTL formulas:

1. $\Diamond \neg$ side_by_side
2. $\Box\, k_e$
3. $\Box k_o \vee \Box\Diamond a_o \wedge \Box\Diamond d_o$.

The synthesis problem, we then are solving, is to find strategies for ego and other which ultimately allow ego to change lane and avoid violating the lower priority objectives. □

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 $\gamma = \upsilon_1\upsilon_2 \ldots \in (2^{V \setminus O})^\omega$ 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 $A \boxdot\!\!\Rightarrow \varphi$.

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 [17] 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 full 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. In addition to full trees, we also consider *partial trees*, which are given as prefix-closed subsets of Υ^*. As partial trees can easily be embedded in full trees (for example, using a labeling $\Upsilon^* \mapsto \mathbb{B}$ that indicates whether a node is present in the partial tree), tree automata can also be used to represent sets of partial trees.

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

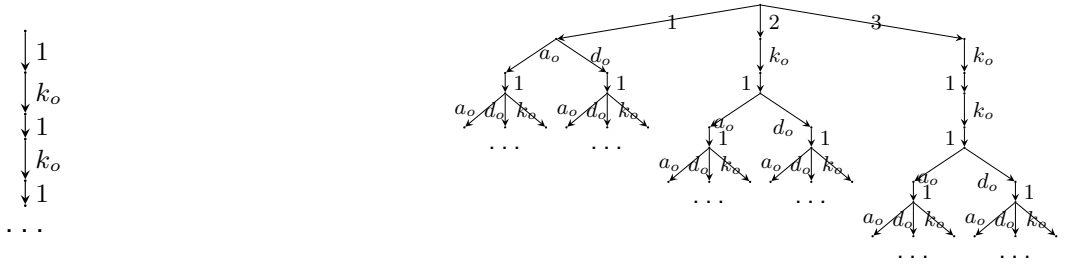
In previous work, we introduced the notion of *remorse-free dominance* [10] in order to deal with situations where it is impossible to achieve the specified objective. Dominance is a weaker version of winning. A strategy $t : (2^I)^* \to 2^O$ *is dominated by* a strategy $s : (2^I)^* \to 2^O$, denoted 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 the objective specification up to priority m, the computation $comp(s, \gamma)$ resulting from s satisfies the objective specification up to priority n and $m \leq n$. 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 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.

Theorem 1. *[11] For a process p, one can construct a parity tree automaton such that the trees accepted by the automaton define exactly the dominant strategies of p.*

Example: In the side by side example, the specification is not admissible: Consider KEEP$^\omega$, a strategy where ego always keeps its speed, i.e., applies k_e^ω. While this would in fact achieve objectives 1 and 2 on p. 6 for ego, if other is accelerating or decelerating at least once, KEEP$^\omega$ cannot beat the alternate strategy ACC, which bets on accelerating in its first move, in the situation where the other car always keeps its speed. By symmetry, ACC is not dominant either.

So, in order to know whether ego can choose KEEP$^\omega$, it needs to know whether other will eventually choose d_o or a_o. If ego chooses to follow KEEP$^\omega$ and other also follows KEEP$^\omega$, ego will not achieve objective 1, $\Diamond \neg side\_by\_side$. Ego will also feel remorse, if it decides to follow strategy ACC but other eventually chooses a_o or d_o, since with KEEP$^\omega$, ego would have achieved objectives 1 and 2. □



a) ego assumes, other does KEEP$^\omega$ b) ego assumes, other guarantees $\neg k_o \vee \bigcirc\neg k_o \vee \bigcirc\bigcirc\neg k_o$

Figure 2: Two assumption trees of ego

5 Computing cooperation assumptions

A component does not necessarily have a dominant strategy, even if the composite system has a dominant strategy. Suppose that, in the example specification $\varphi = (\bigcirc a) \leftrightarrow b$ discussed above, the architecture consists of two processes p and q, with outputs b and a, respectively. Clearly, the system has a dominant, and even winning strategy, e.g., always set a and b to *false*, while p, as discussed above, does not have a dominant strategy. An interesting observation is that p does have a dominant strategy if it is guaranteed that q will set a to *false* in the next step, and that it also has a dominant strategy if it is guaranteed that q will set a to *true* (in which case, p would set b to *true* as well).

We formalize such assumptions on the environment behavior that guarantee the existence of a dominant strategy as *assumption trees*. An *assumption tree* for process p is a subtree of the full $\mathbb{N} \cup 2^{(V \setminus out(p))}$-tree where, on every path, the direction alternates between $\mathbb{N}$ and $2^{V \setminus out(p)}$. Intuitively, each number corresponds to a promise of the environment to a particular restriction on its future behavior. The values of $2^{V \setminus out(p)}$ are then the execution of this promise. The future behavior following different numbers can differ in the inputs provided by the environment at specific points in time in the future and also in the number of branchings with natural numbers that occur in the future.

Example: In our side by side example, a very restrictive assumption expresses that *"other always chooses k_o"*. Its assumption tree can be envisioned as a tree that degenerates to a simple linear graph as in Fig. 2 a). At the root the single edge announces that only one future is assumed initially. From there the single edge labeled k_o encodes that the only allowed first step of this future is k_o. Likewise, the single edge at node $1\,k_o$ announces that only one future development is distinguished at that point. The single edge at $1\,k_o\,1$ encodes that the next step has to be k_o again, and so on. With respect to this assumption, ego cannot achieve $\Diamond\neg side\_by\_side \wedge \Box k_e$, but ego can achieve objective 1 without remorse following any strategy that chooses at least once a_e or d_e. Any such strategy is dominant wrt. the assumption tree in Fig. 2 a).

Ego may also assume that *"other eventually chooses $\neg k_o$"*, where $\neg k_o$ abbreviates $a_o \vee d_o$. The strategy KEEP$^\omega$ is the only dominant strategy with respect to this assumption and any stronger assumption – including the one depicted in Fig. 2 b). The tree of Fig. 2 b) roughly describes ego's assumption that other changes its speed within the first three steps. At the tree's root, three assumptions on other's behavior are distinguished. The first, at node 1, describes that other chooses initially $\neg k_o$. The subtrees at $1\,a_o$ and $1\,d_o$, respectively, both describe the assumption that all future development is possible: The outgoing single edge at node $1\,a_o$ announces that then only a single future is distinguished. The three outgoing egdes at $1\,a_o\,1$ specify that this future assumption allows to apply any acceleration, i.e., a_o,

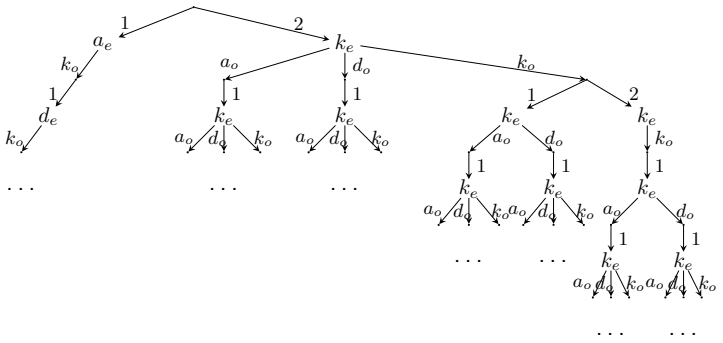


Figure 3: Side_by_side example: A strategy annotated assumption tree of ego.

d_o or k_o, as second step. From there again, but not depicted, a single edge leads to a node with three outgoing edges labeled a_o, d_o and k_o denoting that any choice for other's acceleration is accepted for this step as well; and so on. The subtree at node 2 in Fig. 2 b) describes that other initially chooses k_o, then $\neg k_o$ and thereafter the future is unconstrained. The subtree at node 3 in Fig. 2 b) describes that other is assumed to choose $k_o\,k_o$ followed by $\neg k_o$ and then the future is no further constrained.

For an example of an assumption tree with infinite branching at the root, consider ego's assumption that other announces in subtree i to do $\neg k_o$ at step $i \in \mathbb{N}$. □

To ensure that a given assumption tree guarantees the existence of a dominant strategy, we annotate each node that is reached by a natural number with an output value from $2^{out(p)}$. One such strategy annotation t is *dominated* by another strategy annotation s of the same assumption tree, denoted by $t \preceq s$, iff for every alternating sequence of natural numbers and environment outputs from $2^{out(p)}$ that is present in the tree, if the computation resulting from t satisfies the objective specification up to priority m, then the computation resulting from s satisfies the objective specification up to priority n and $m \leq n$. A strategy annotation s is *dominant* iff, for all strategies t, $t \preceq s$. We say that an assumption tree is *dominant* (and it *guarantees the existence of a dominant strategy*) iff it has a dominant strategy annotation.

Example: In Fig. 2 b) where the assumption tree basically expresses that other guarantees to choose $\neg k_o$ initially or within the next two steps, we can annotate the assumption tree with strategy KEEP$^\omega$ to get a dominant strategy annotation.

An example of a strategy annotated assumption tree is shown in Fig. 3. There we basically combined the assumptions *"other chooses always k_o"* and *"other guarantees to choose $\neg k_o$ initially or within its next two steps"* as subtrees 1 and 2, respectively. The strategy annotation results from applying the strategy that alternates a_e and d_e, $(a_e d_e)^\omega$ on subtree 1, and the KEEP$^\omega$ strategy on subtree 2. □

We now describe the computation of a tree automaton that recognizes the set of all assumption trees for a given process with a dominant annotation. A small technical difficulty is that assumption trees have an infinite branching degree. In order to represent the set of assumption trees as a tree automaton, we use the standard unary representation of $\mathbb{N}^*$ in $\mathbb{B}^*$, where a sequence $abc\ldots \in \mathbb{N}^*$ is encoded as $0^a 1 0^b 1 0^c 1 \ldots$. The construction then proceeds in the following steps. (1) We build a word automaton $\mathcal{A}$ that checks that no alternative would do better on the present path. (2) We build a tree automaton $\mathcal{B}$ that reads in a strategy annotated assumption tree and uses $\mathcal{A}$ to verify that this annotation is dominant.

Theorem 2. *For a single process p, one can construct a parity tree automaton such that the trees accepted by the automaton are the dominant strategy annotated assumption trees of p.*

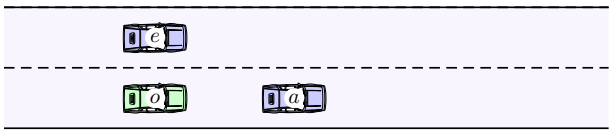


Figure 4: Initial scenario for the incremental synthesis: Three cars on a freeway.

6 Incremental synthesis of cooperation strategies

We now use the construction of the environment assumptions from Section 5 to incrementally synthesize a distributed system by propagating the assumptions. We assume that the processes $P = \{p_1, p_2, \ldots, p_n\}$ are ordered $p_1 > p_2 > \ldots p_n$ according to criticality. We begin by computing the set A_1 of strategy annotated assumption trees for process p_1. Next, we compute the set A_2 of strategy annotated assumption trees for process p_2 that additionally satisfy A_1. We continue up to process p_n.

Theorem 3. *For an architecture with a set $P = \{p_1, p_2, \ldots, p_n\}$ of processes, ordered $p_1 > p_2 > \ldots p_n$ according to criticality, one can construct, for each process $p_i, i = 1..n$, a parity tree automaton, such that the trees accepted by the automaton are assumption trees that guarantee the existence of dominant strategies for all p_j with $j \leq i$.*

The set of assumption trees constructed in this way represent the remaining assumptions on the external inputs. Since we cannot constrain the external environment, we find the final distributed implementation by selecting some assumption tree that allows for all possible future environment behaviors.

Example: Let us consider three cars as in Fig. 4. Ego and other are initially side by side and the car ahead starts in front of other. The synthesis problem is to find a strategy for ego, other and ahead that satisfies the prioritized specification given by the following objectives listed in order of decreasing priority.

1. $\Diamond \neg side\_by\_side_e \wedge \Box(curve \Rightarrow \bigcirc\bigcirc \neg a_a)$
2. $\Box k_e$
3. $\Box(\bigcirc d_a \Rightarrow d_o) \wedge \Box(\bigcirc k_a \Rightarrow \neg a_o)$

The objectives for ego remain the same. ahead is required not to accelerate in a bend, i.e., when ahead notices a curve two steps ahead, it may not accelerate two steps later, when in the bend. The car other has to ensure not to get faster than ahead. This is here expressed via a sufficient constraint on its acceleration: other is never allowed to accelerate stronger than ahead.

We order processes ego > other > ahead and start the incremental synthesis with ego. We have already seen elements of the set of ego's assumption trees for the simpler setting of two cars in Sect. 5. An assumption tree for the three car example will additionally be labeled with acceleration values for ahead and the environmental input announcing bends $curve\_ahead \in \{curve, no\_curve\}$. But since ego depends on neither of them, they are of no relevance for the assumption tree, i.e., for each assumption tree in the two car setting there will be now an assumption tree where ahead and curve_ahead are unconstrained.

In the following we will illustrate the assumption propagation, starting with the exemplary assumption tree of ego depicted in Fig. 3. A strategy annotated assumption tree of other, saat(other), is given in Fig. 5. It describes the assumption that ahead chooses a_a at least every other time. For the sake of a simple illustration, we have chosen a strong assumption, so that we can annotate it with a simple strategy. $(d_o a_o)^\omega$ is dominant strategy wrt. the assumption tree in Fig. 5.

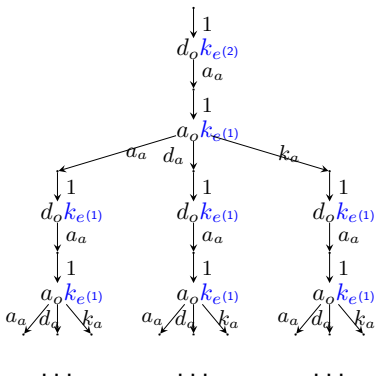


Figure 5: A strategy annotated assumption tree of other: ahead is assumed to apply a_a at least every other time. Other's dominant strategy is to alternate between d_o and a_o. The tree satisfies saat(ego) of Fig. 3.

Now, the saat(other) satisfies ego's assumption *"other guarantees to choose $\neg k_o$ within its next two steps"*. Intuitively, first other chooses a dominant strategy for the given future assumption on ahead. Here other decides to apply $(d_o a_o)^\omega$ given the assumption of Fig. 5. Then a compatible dominant strategy for ego is determined based on the propagated strategy annotated assumption trees. The match with ego's strategy is highlighted in blue in Fig. 5. Now, nodes are also annotated with ego's acceleration choices. The numbers in brackets refer to the respective edge labels in the assumption tree in Fig. 3.

Intuitively, the weakest dominant assumption for other is *"ahead announces when it will do d_a and when it will do k_a next"* and a dominant strategy for other is, e.g., to choose d_o, if d_a or k_a is announced and other chooses a_o, if ahead does not announce either d_a or k_a.

At this point the strategy annotated assumption tree has inputs curve_ahead and the acceleration of ahead. We again omitted in Fig. 5 variables at the edge annotation that do not influence the strategy.

Next and final element according to our process order is ahead. So we choose a dominant strategy for ahead, for instance a strategy that chooses d_a in two steps time, in case the environment announces a bend, otherwise it chooses to accelerate. The strategies for ego and ahead are now determined based on the propagated (ego,other)-strategy annotated assumption trees, as illustrated in the previous step. Note, that ahead is informed of a bend two steps in advance. Intuitively, ahead gets informed that a bend is ahead, it then cooperates with other by choosing an appropriate edge for other in its strategy annotated assumption tree, i.e., it announces its decision to decelerate in two steps time, then other decelerates and at the next step ahead decelerates while in the bend. □

7 Conclusion

Relying on cyber-physical systems increases our vulnerability towards technical failures caused by unpredicted emergent behavior. Understanding what pieces of information must be exchanged with whom, understanding what level of cooperation is necessary for the overall objectives –topics addressed in this paper– constitute initial steps towards a rigorous design discipline for such systems. They also expose the vulnerability of the system towards intruders: the analysis clearly can be used to highlight the chaos, which can be inserted into the system by pretending to pass information and then not acting accordingly.

The paper thus both provides an initial step towards cooperatively establishing systems objectives, as well as exposing its vulnerabilities. We will explore both avenues in our future research in the application domain of cooperative driver assistance systems.

References

[1] Romain Brenguier, Jean-François Raskin & Ocan Sankur (2015): *Assume-Admissible Synthesis*. In Luca Aceto & David de Frutos Escrig, editors: *26th International Conference on Concurrency Theory (CONCUR 2015)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 42, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 100–113.

[2] J. Richard Büchi & Lawrence H. Landweber (1990): *Solving Sequential Conditions by Finite-State Strategies*. In Saunders Mac Lane & Dirk Siefkes, editors: *The Collected Works of J. Richard Büchi*, Springer New York, pp. 525–541.

[3] Krishnendu Chatterjee & Thomas A. Henzinger (2007): *Assume-Guarantee Synthesis*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 4424, Springer Berlin Heidelberg, pp. 261–275.

[4] Krishnendu Chatterjee, Thomas A. Henzinger & Barbara Jobstmann (2008): *Environment Assumptions for Synthesis*, pp. 147–161. Springer Berlin Heidelberg, Berlin, Heidelberg.

[5] Alonzo Church (1963): *Logic, Arithmetic and Automata*. In: *Proc. 1962 Intl. Congr. Math.*, Upsala, pp. 23–25.

[6] S. Coogan & M. Arcak (2014): *Freeway traffic control from linear temporal logic specifications*. In: *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pp. 36–47.

[7] W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann & B. Wirtz (2006): *Automatic verification of hybrid systems with large discrete state space*. In: *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis*, *Lecture Notes in Computer Science* 4218, Springer-Verlag.

[8] W. Damm, G. Pinto & S. Ratschan (2007): *Guaranteed Termination in the Verification of LTL Properties of Non-linear Robust Discrete Time Hybrid Systems*. *Int. Journal of Foundations of Computer Science* 18(1), pp. 63–86.

[9] Werner Damm, Stefan Disch, Hardi Hungar, Swen Jacobs, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann & Boris Wirtz (2007): *Exact State Set Representations in the Verification of Linear Hybrid Systems with Large Discrete State Space*. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino & Yoshio Okamura, editors: *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007*, *Lecture Notes in Computer Science* 4762, Springer, Tokyo, Japan, pp. 425–440.

[10] Werner Damm & Bernd Finkbeiner (2011): *Does It Pay to Extend the Perimeter of a World Model?* In Michael Butler & Wolfram Schulte, editors: *FM*, *Lecture Notes in Computer Science* 6664, Springer, pp. 12–26.

[11] Werner Damm & Bernd Finkbeiner (2014): *Automatic Compositional Synthesis of Distributed Systems*. In Cliff Jones, Pekka Pihlajasaari & Jun Sun, editors: *FM 2014: Formal Methods*, *Lecture Notes in Computer Science* 8442, Springer International Publishing, pp. 179–193.

[12] Werner Damm, Hans-Jörg Peter, Jan Rakow & Bernd Westphal (2013): *Can we build it: formal synthesis of control strategies for cooperative driver assistance systems*. *Mathematical Structures in Computer Science* 23, pp. 676–725.

[13] Tomáš Dzetkulič & Stefan Ratschan (2011): *Incremental Computation of Succinct Abstractions for Hybrid Systems*. In Uli Fahrenberg & Stavros Tripakis, editors: *Formal Modeling and Analysis of Timed Systems*, *Lecture Notes in Computer Science* 6919, Springer Berlin Heidelberg, pp. 271–285.

[14] Christian Frese (2010): *A Comparison of Algorithms for Planning Cooperative Motions of Cognitive Automobiles*. Technical Report IES-2010-06, Lehrstuhl fr Interaktive Echtzeitsysteme.

[15] Christian Frese & Jrgen Beyerer (2010): *Planning Cooperative Motions of Cognitive Automobiles Using Tree Search Algorithms*. In Rdiger Dillmann, Jrgen Beyerer, UweD. Hanebeck & Tanja Schultz, editors: *KI 2010: Advances in Artificial Intelligence*, *Lecture Notes in Computer Science* 6359, Springer Berlin Heidelberg, pp. 91–98.

[16] Paul Gastin, Nathalie Sznajder & Marc Zeitoun (2006): *Distributed Synthesis for Well-Connected Architectures*. In S. Arun-Kumar & Naveen Garg, editors: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science* 4337, Springer Berlin Heidelberg, pp. 321–332.

[17] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games*. *LNCS* 2500, Springer-Verlag.

[18] L. Liu, D. Sheridan, V. Athavale & S. Vasudevan (2011): *Automatic generation of assertions from system level design using data mining*. In: *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pp. 191–200.

[19] John Lygeros & Nancy Lynch (1998): *Strings of vehicles: Modeling and safety conditions*. In ThomasA. Henzinger & Shankar Sastry, editors: *Hybrid Systems: Computation and Control*, *Lecture Notes in Computer Science* 1386, Springer Berlin Heidelberg, pp. 273–288.

[20] John Lygeros, Claire Tomlin & Shankar Sastry (1997): *Multiobjective hybrid controller synthesis*. In Oded Maler, editor: *Hybrid and Real-Time Systems*, *Lecture Notes in Computer Science* 1201, Springer Berlin Heidelberg, pp. 109–123.

[21] Oscar Mickelin, Necmiye Ozay & Richard M. Murray (2014): *Synthesis of Correct-by-construction Control Protocols for Hybrid Systems Using Partial State Information*. Available at `http://www.cds.caltech.edu/~murray/papers/mom14-acc.html`. American Control Conference (ACC).

[22] A. Pnueli & R. Rosner (1989): *On the Synthesis of a Reactive Module*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, ACM, New York, NY, USA, pp. 179–190.

[23] A. Pnueli & R. Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, SFCS '90, IEEE Computer Society, Washington, DC, USA, pp. 746–757 vol.2.

[24] M.O. Rabin (1972): *Automata on Infinite Objects and Church's Problem*. Conference Board of the mathematical science, Conference Board of the Mathematical Sciences. Available at `http://books.google.de/books?id=AY5xCRs_-bQC`.

[25] A. Richards & J.P. How (2002): *Aircraft trajectory planning with collision avoidance using mixed integer linear programming*. In: *American Control Conference, 2002. Proceedings of the 2002*, 3, pp. 1936–1941 vol.3.

[26] C. Tomlin, G. J. Pappas, J. Kosecka, J. Lygeros & S. S. Sastry (1998): *Advanced air traffic automation: A case study in distributed decentralized control*. In: *Control Problems in Robotics and Automation*, Springer-Verlag, pp. 261–295.

[27] T. Wongpiromsarn, U. Topcu & R.M. Murray (2012): *Receding Horizon Temporal Logic Planning*. *Automatic Control, IEEE Transactions on* 57(11), pp. 2817–2830.

[28] Tichakorn Wongpiromsarn, Ufuk Topcu & Richard M. Murray (2013): *Synthesis of Control Protocols for Autonomous Systems*. *Unmanned Systems* 01(01), pp. 21–39.