LTL Reactive Synthesis with a Few Hints

Mrudula Balachander[1](✉), Emmanuel Filiot, and Jean-François Raskin

Université libre de Bruxelles, Brussels, Belgium
mbalacha@ulb.be

Abstract. We study a variant of the problem of synthesizing Mealy machines that enforce LTL specifications against all possible behaviours of the environment, including hostile ones. In the variant studied here, the user provides the high level LTL specification φ of the system to design, and a set E of examples of executions that the solution must produce. Our synthesis algorithm first generalizes the user-provided examples in E using tailored extensions of automata learning algorithms, while preserving realizability of φ. Second, it turns the (usually) incomplete Mealy machine obtained by the learning phase into a complete Mealy machine realizing φ. The examples are used to guide the synthesis procedure. We prove learnability guarantees of our algorithm and prove that our problem, while generalizing the classical LTL synthesis problem, matches its worst-case complexity. The additional cost of learning from E is even polynomial in the size of E and in the size of a symbolic representation of solutions that realize φ, computed by the synthesis tool Acacia-Bonzai. We illustrate the practical interest of our approach on a set of examples.

1 Introduction

Reactive systems are notoriously difficult to design and even to specify correctly [1,13]. As a consequence, formal methods have emerged as useful tools to help designers to built reactive systems that are correct. For instance, model-checking asks the designer to provide a model, in the form of a Mealy machine $\mathcal{M}$, that describes the reactions of the system to events generated by its environment, together with a description of the *core correctness properties* that must be enforced. Those properties are expressed in a logical formalism, typically as an LTL formula φ_{CORE}. Then an algorithm decides if $\mathcal{M} \models \varphi_{\mathsf{CORE}}$, i.e. if all executions of the system in its environment satisfy the specification. Automatic reactive synthesis is more ambitious: it aims at automatically generating a model from a high level description of the "*what*" needs to be done instead of the "*how*" it has to be done. Thus the user is only required to provide an LTL specification φ and the algorithm automatically generates a Mealy machine $\mathcal{M}$ such that $\mathcal{M} \models \varphi$ whenever φ is *realizable*. Unfortunately, it is most of the time not sufficient to provide the core correctness properties φ_{CORE} to obtain a Mealy machine $\mathcal{M}$ that is useful in practice, as illustrated next.

Example 1. [Synthesis from φ_{CORE} - Mutual exclusion] Let us consider the classical problem of *mutual exclusion*. In the simplest form of this problem, we need to design an arbiter that receives requests from two processes, modeled

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 309–328, 2023.
https://doi.org/10.1007/978-3-031-30820-8_20

by two atomic propositions r_1 and r_2 controlled by the environment, and that grants accesses to the critical section, modeled as two atomic propositions g_1 and g_2 controlled by the system. The core correctness properties (the *what*) are: (i) mutual access, i.e. it is never the case that the access is granted to both processes at the same time, (ii) fairness, i.e. processes that have requested access eventually get access to the critical section. These core correctness specifications for mutual exclusion (ME) are easily expressed in LTL as follows: $\varphi^{\mathsf{ME}}_{\mathsf{CORE}} \equiv \Box(\neg g_1 \vee \neg g_2) \wedge \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2)$. Indeed, this formula expresses the core correctness properties that we would model check no matter *how* $\mathcal{M}$ implements mutual exclusion, e.g. Peterson, Dedekker, Backery algorithms, etc. Unfortunately, if we submit $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ to an LTL synthesis procedure, implemented in tools like Acacia-Bonzai [11], BoSy [17], or Strix [25], we get the solution $\mathcal{M}$ depicted in 1-(left) (all three tools return this solution). While this solution is perfectly correct and realizes the specification $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$, the solution ignores the inputs from the environment and grants access to the critical sections in a round robin fashion. Arguably, it may not be considered as an efficient solution to the mutual exclusion problem. This illustrates the limits of the synthesis algorithm to solve the design problem by providing *only* the core correctness specification of the problem, i.e. the *what*, only. To produce useful solutions to the mutual exclusion problem, more guidance must be provided.

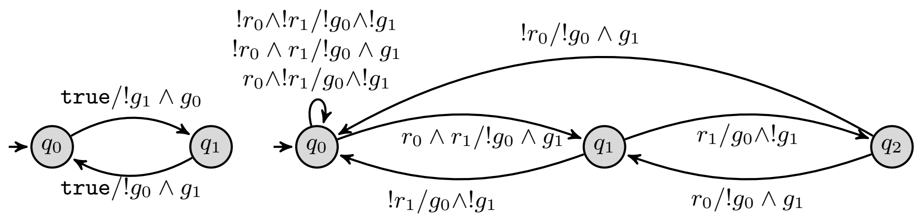

Fig. 1: (Left) The solution of Strix to the mutual exclusion problem for high level specification $\varphi^{\mathsf{ME}}_{LOW}$. Edge labels are of the form φ/ψ where φ: Boolean formula on input atomic propositions (Boolean variables controlled by environment) and ψ: maximally consistent conjunction of literals over set of output propositions (Boolean variables controlled by system). (Right) A natural solution that could be drawn by hand, and is automatically produced by our learning/synthesis algorithm for the same specification plus with two simple examples.

The main question is now: *how should we specify these additional properties ?* Obviously, if we want to use the "plain" LTL synthesis algorithm, there is no choice: we need to reinforce the specification $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ with additional lower level properties $\varphi^{\mathsf{ME}}_{\mathsf{LOW}}$. Let us go back to our running example.

Example 2. [Synthesis from $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ and $\varphi^{\mathsf{ME}}_{\mathsf{LOW}}$] To avoid solutions with *unsolicited grants*, we need to reinforce the core specification. The Strix online demo website proposes to add the following 3 LTL formulas $\varphi^{\mathsf{ME}}_{\mathsf{LOW}}$ to $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ (see Full arbitrer $n = 2$, at `https://meyerphi.github.io/strix-demo/`): (1) $\bigwedge_{i\in\{1,2\}} \Box((g_i \wedge \Box\neg r_i) \rightarrow \Diamond\neg g_i)$, (2) $\bigwedge_{i\in\{1,2\}} \Box(g_i \wedge \bigcirc(\neg r_i \wedge \neg g_i) \rightarrow \bigcirc(r_i \mathsf{R} \neg g_i))$, and (3) $\bigwedge_{i\in\{1,2\}}(r_i \mathsf{R} \neg g_i)$. Strix, on the specification $\varphi^{\mathsf{ME}}_{\mathsf{CORE}} \wedge \varphi^{\mathsf{ME}}_{\mathsf{LOW}}$, provides us with a better solution, but it is more complex than needed (it has 9 states: refer [5]) and clearly does not look like an optimal solution to our mutual exclusion problem. E.g., the model of Fig. 1-(right) is arguably more natural. How can we get this model without coding it into the LTL specification, which would diminish greatly the interest of using a synthesis procedure in the first place?

In general, higher level properties are properties that need to be met by all implementations, e.g. safety-critical properties. In contrast, lower level properties are more about a specific implementation, its expected behaviour and efficiency. At this point, it is legitimate to question the adequacy of LTL as a specification language for *lower level* properties, and so as a way to guide the synthesis procedure towards relevant solutions to realize φ_{CORE}. In this paper, we introduce an alternative to guide synthesis toward useful solutions that realize φ_{CORE}: we propose to use examples of executions that illustrate behaviors of expected solutions. We then restrict the search to solutions that *generalize* those examples. Examples, or scenarios of executions, are accepted in requirement engineering as an adequate tool to elicit requirements about complex systems [12]. For reactive system design, examples are particularly well-suited as they are usually much easier to formulate than full blown solutions, or even partial solutions. It is because, when formulating examples, the user controls *both* the inputs *and* the outputs, avoiding the main difficulty of reactive system design: having to cope with *all* possible environment inputs. We illustrate this on our running example.

Example 3. [Synthesis from $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ and examples] Let us keep, as the LTL specification, $\varphi^{\mathsf{ME}}_{\mathsf{CORE}}$ only, and let us consider the following simple prefix of executions that illustrate how solutions to mutual exclusion should behave:
(1) $\{!r_1, !r_2\}.\{!g_1, !g_2\}\#\{r_1, !r_2\}.\{g_1, !g_2\}\#\{!r_1, r_2\}.\{!g_1, g_2\}$
(2) $\{r_1, r_2\}.\{g_1, !g_2\}\#\{!r_1, !r_2\}.\{!g_1, g_2\}$
These trace prefixes prescribe reactions to typical *fixed* finite input sequences: (1) if there is no request initially, then no access is granted (note that this excludes already the round robin solution), if process 1 and 2 request subsequently, process 1 is granted first and then process 2 is granted after, (2) if both process request simultaneously, then process 1 is granted first and then process 2 is granted after. Given those two simple traces together with φ_{CORE}, our algorithm generates the solution of Fig. 1-(right). Arguably, the solution is now simple and natural.

Contributions First, we provide a synthesis algorithm SynthLearn that, given an LTL specification φ_{CORE} and a finite set E of prefixes of executions, returns a Mealy machine $\mathcal{M}$ such that $\mathcal{M} \models \varphi_{\mathsf{CORE}}$, i.e. $\mathcal{M}$ realizes φ_{CORE}, and $E \subseteq \mathsf{Prefix}(L(\mathcal{M}))$, i.e. $\mathcal{M}$ is compatible with the examples in E, if such a machine $\mathcal{M}$ exists. It returns *unrealizable* otherwise. Additionally, we require SynthLearn to *generalize* the decisions illustrated in E. This learnability requirement is usually formalized in automata learning with a *completeness criterium* that we adapt here as follows: for all specifications φ_{CORE}, and for all Mealy machines $\mathcal{M}$ such that $\mathcal{M} \models \varphi_{\mathsf{CORE}}$, there is a small set of examples E (polynomial in $|\mathcal{M}|$) such that $L(\text{SynthLearn}(\varphi_{\mathsf{CORE}}, E)) = L(\mathcal{M})$. We prove this completeness result in Theorem 4 for safety specifications and extend it to ω-regular and LTL specifications in Section 4, by reduction to safety.

Second, we prove that the worst-case execution time of SynthLearn is 2ExpTime (Theorem 7), and this is worst-case optimal as the plain LTL synthesis problem (when $E = \emptyset$) is already known to be 2ExpTime-Complete [27]. SynthLearn first *generalizes* the examples provided by the user while maintaining realizability of φ_{CORE}. This generalization leads to a Mealy machine with possibly missing transitions (called a preMealy machine). Then, this preMealy machine is extended into a (full) Mealy machine that realizes φ_{CORE} against all behaviors of the environment. During the completion phase, SynthLearn reuses as much as possible decisions that have been generalized from the examples. The generalization phase is essential to get the most out of the examples. Running classical synthesis algorithms on $\varphi_{\mathsf{CORE}} \wedge \varphi_E$, where φ_E is an LTL encoding of E, often leads to more complex machines that fail to generalize the decisions taken along the examples in E. While the overall complexity of SynthLearn is 2ExpTime and optimal, we show that it is only polynomial in the size of E and in a well-chosen symbolic representation a set of Mealy machines that realize φ_{CORE}, see Theorem 6. This symbolic representation takes the form of an antichain of functions and tends to be compact in practice [19]. It is computed by default when Acacia-Bonzai is solving the plain LTL synthesis problem of φ_{CORE}. So, generalizing examples while maintaining realizability only comes at a marginal polynomial cost. We have implemented our synthesis algorithm in a prototype, which uses Acacia-Bonzai to compute the symbolic antichain representation. We report on the results we obtain on several examples.

Related works Scenarios of executions have been advocated by researchers in requirements engineering to elicit specifications, see e.g. [12,14] and references therein. In [28], learning techniques are used to transform examples into LTL formulas that generalize them. Those methods are complementary to our work, as they can be used to obtain the high level specification φ_{CORE}.

In non-vacuous synthesis [8], examples are added automatically to an LTL specification in order to force the synthesis procedure to generate solutions that are non-vacuous in the sense of [23]. The examples are generated directly from the syntax of the LTL specification and they cannot be proposed by the user. This makes our approach and this approach orthogonal and complementary. Indeed, we could use the examples generated automatically by the non-vacuous approach and ask the user to validate them as desirable or not. Our method is more flexible, it is semi-automatic and user centric: the user can provide any example he/she likes and so it offers more flexibility to drive the synthesis procedure to solutions that the user deems as interesting. Furthermore, our synthesis procedure is based on learning algorithms, while the algorithm in [8] is based on constraint solving and does not offer guarantees of generalization, unlike our algorithm (see Thm 4).

Supplementing the formal specification with additional user-provided information is at the core of the *syntax-guided synthesis* framework (SyGuS [3]), implemented for instance in *program by sketching* [31]: in SyGuS, the specification is a logical formula and candidate programs are syntactically restricted by a user-provided grammar, to limit and guide the search. The search is done by using counter-example guided inductive synthesis techniques (CEGIS) which rely on learning [32]. In contrast to our approach, examples are not user-provided but automatically generated by model-checking the candidate programs against the specification. The techniques are also orthogonal to ours: SyGuS targets programs syntactically defined by expressions over a decidable background theory, and heavily relies on SAT/SMT solvers. Using examples to synthesise programs (*programming by example*) has been for instance explored in the context of string processing programs for spreadsheets, based on learning [30], and is a current trend in AI (see for example [26] and the citations therein). However this approach only relies on examples and not on logical specifications.

[4] explores the use of formal specifications and scenarios to synthesize distributed protocols. Their approach also follows two phases: first, an incomplete machine is built from the scenarios and second, it is turned into a complete one. But there are two important differences with our work. First, their first phase does not rely on learning techniques and does not try to generalize the provided examples. Second, in their setting, all actions are controllable and there is no adversarial environment, so they are solving a satisfiability problem and not a realizability problem as in our case. Their problem is thus computationally less demanding than the problem we solve: PSPACE versus 2EXPTIME for LTL specs.

The synthesis problem targeted in this paper extends the LTL synthesis problem. Modern solutions for this problem use automata constructions that avoid Safra's construction as first proposed in [24], and simplified in [29,18], and more recently in [16]. Efficient implementations of Safraless constructions are available, see e.g. [9,17,25,15]. Several previous works have proposed alternative approaches to improve on the quality of solutions that synthesis algorithms can offer. A popular research direction, orthogonal and complementary to the one proposed here, is to extend the formal specification with quantitative aspects, see e.g. [6,10,22,2], and only synthesize solutions that are optimal.

The first phase of our algorithm is inspired by automata learning techniques based on state merging algorithms like RPNI [21,20]. Those learning algorithms need to be modified carefully to generate partial solutions that preserve realizability of φ_{CORE}. Proving completeness as well as termination of the completion phase in this context requires particular care.

2 Preliminaries on the reactive synthesis problem

Words, languages and automata An alphabet is a finite set of symbols. A *word* u (resp. ω-word) over an alphabet Σ is a finite (resp. infinite sequence) of symbols from Σ. We write ϵ for the empty word, and denote by $|u| \in \mathbb{N} \cup \{\infty\}$ the length of u. In particular, $|\epsilon| = 0$. For $1 \leq i \leq j \leq |u|$, we let $u[i{:}j]$ be the infix of u from position i to position j, both included, and write $u[i]$ instead of $u[i{:}i]$. The set of finite (resp. ω-) words over Σ is denoted by Σ^* (resp. Σ^ω). We let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Given two words $u \in \Sigma^*$ and $v \in \Sigma^\infty$, u is a *prefix* of v, written $u \preceq v$, if $v = uw$ for some $w \in \Sigma^\infty$. The set of prefixes of v is denoted by $\mathsf{Prefs}(v)$. Finite words are linearly ordered according to the length-lexicographic order $\preceq_{ll}$, assuming a linear order $<_\Sigma$ over Σ: $u \preceq_{ll} v$ if $|u| < |v|$ or $|u| = |v|$ and $u = p\sigma_1 u'$, $v = p\sigma_2 v'$ for some $p, u', v' \in \Sigma^*$ and some $\sigma_1 <_\Sigma \sigma_2$. In this paper, whenever we refer to the order $\preceq_{ll}$ for words over some alphabet, we implicitly assume the existence of an arbitrary linear order over that alphabet. A *language* (resp. ω-language) over an alphabet Σ is a subset $L \subseteq \Sigma^*$ (resp. $L \subseteq \Sigma^\omega$).

In this paper, we fix two alphabets $\mathcal{I}$ and $\mathcal{O}$ whose elements are called inputs and outputs respectively. Given a word $u \in (\mathcal{I}\mathcal{O})^\infty$, we let $\mathsf{in}(u) \in \mathcal{I}^\infty$ be the word obtained by erasing all $\mathcal{O}$-symbols from u. We define $\mathsf{out}(u)$ similarly and naturally extend both functions to languages.

Automata over ω-words A *parity automaton* is a tuple $\mathcal{A} = (Q, Q_{\mathsf{init}}, \Sigma, \delta, d)$ where Q is a finite non empty set of states, $Q_{\mathsf{init}} \subseteq Q$ is a set of initial states, Σ is a finite non empty alphabet, $\delta : Q \times \Sigma \to 2^Q \setminus \{\emptyset\}$ is the transition function, and $d : Q \to \mathbb{N}$ is a parity function. The automaton $\mathcal{A}$ is *deterministic* when $|Q_{\mathsf{init}}| = 1$ and $|\delta(q, \sigma)| = 1$ for all $q \in Q$. The transition function is extended naturally into a function $\mathsf{Post}^* : Q \times \Sigma^* \to 2^Q \setminus \{\emptyset\}$ inductively as follows: $\mathsf{Post}^*(q, \epsilon) = \{q\}$ for all $q \in Q$ and for all $(u, \sigma) \in \Sigma^* \times \Sigma$, $\mathsf{Post}^*(q, u\sigma) = \bigcup_{q' \in \mathsf{Post}^*(q,u)} \delta(q', \sigma)$.

A run of $\mathcal{A}$ on an ω-word $w = w_0 w_1 \dots$ is an infinite sequence of states $r = q_0 q_1 \dots$ such that $q_0 \in Q_{\mathsf{init}}$, and for all $i \in \mathbb{N}$, $q_{i+1} \in \delta(q_i, w_i)$. The run r is said to be *accepting* if the minimal colour it visits infinitely often is even, i.e. $\liminf(d(q_i))_{i \geq 0}$ is *even*. We say that $\mathcal{A}$ is a *Büchi automaton* when $\mathsf{dom}(d) = \{0, 1\}$ (1-coloured states are called accepting states), a *co-Büchi automaton* when $\mathsf{dom}(d) = \{1, 2\}$, a *safety automaton* if it is a Büchi automaton such that the set of 1-coloured states, called *unsafe states* and denoted Q_{usf}, forms a *trap*: for all $q \in Q_{\mathsf{usf}}$, for all $\sigma \in \Sigma$, $\delta(q, \sigma) \subseteq Q_{\mathsf{usf}}$, and a *reachability automaton* if it is $\{0, 1\}$-coloured and the set of 0-coloured states forms a trap.

Finally, we consider the existential and universal interpretations of nondeterminism: under the *existential (resp. universal) interpretation*, a word $w \in \Sigma^\omega$ is in the language of $\mathcal{A}$, if there exists a run r on w such that r is accepting (resp. for all runs r on w, r is accepting). We denote the two languages defined by these two interpretations $L^\exists(\mathcal{A})$ and $L^\forall(\mathcal{A})$ respectively. Note that if $\mathcal{A}$ is deterministic, then the existential and universal interpretations agree, and we write $L(\mathcal{A})$ for $L^\forall(\mathcal{A}) = L^\exists(\mathcal{A})$. For a deterministic automaton $\mathcal{A}$, the initial state is fixed to the singleton $\{q\}$.

For a *co-Büchi automaton*, we also define a strengthening of the acceptance condition, called K-co-Büchi, which requires, for $K \in \mathbb{N}$, that a run visits at most K times a state labelled with 1 to be accepting. Formally, a run $r = q_0 q_1 \dots q_n \dots$ is *accepting* for the K-co-Büchi acceptance condition if $|\{i \geq 0 \mid d(q_i)) = 1\}| \leq K$. The language defined by $\mathcal{A}$ for the K-co-Büchi acceptance condition and universal interpretation is denoted by $L^\forall_K(\mathcal{A})$. Note that this language is a *safety* language because if a prefix of a word $p \in \Sigma^*$ is such that $\mathcal{A}$ has a run prefix on p that visits more than K times a states labelled with color 1, then all possible extensions $w \in \Sigma^\omega$ of p are rejected by $\mathcal{A}$.

(Pre)Mealy machines Given a (partial) function f from a set X to a set Y, we denote by $\mathsf{dom}(f)$ its domain, i.e. the of elements $x \in X$ such that $f(x)$ is defined. A *preMealy machine* $\mathcal{M}$ on an input alphabet $\mathcal{I}$ and output alphabet

$\mathcal{O}$ is a triple $(M, m_{\mathsf{init}}, \Delta)$ such that M is a non-empty set of states, $m_{\mathsf{init}} \in M$ is the initial state, $\Delta : Q \times \mathcal{I} \to \mathcal{O} \times M$ is a partial function. A pair (m, i) is a hole in $\mathcal{M}$ if $(m, \mathsf{i}) \notin \mathsf{dom}(\Delta)$. A *Mealy machine* is a preMealy machine such that Δ is total, i.e., $\mathsf{dom}(\Delta) = M \times \mathcal{I}$.

We define two semantics of a preMealy machine $\mathcal{M} = (M, m_{\mathsf{init}}, \Delta)$ in terms of the languages of finite and infinite words over $\mathcal{I} \cup \mathcal{O}$ they define. First, we define two (possibly partial functions) $\mathsf{Post}_{\mathcal{M}} : M \times \mathcal{I} \to M$ and $\mathsf{Out}_{\mathcal{M}} : M \times \mathcal{I} \to \mathcal{O}$ such that $\Delta(m, \mathsf{i}) = (\mathsf{Post}_{\mathcal{M}}(m, \mathsf{i}), \mathsf{Out}_{\mathcal{M}}(m, \mathsf{i}))$ for all $(m, \mathsf{i}) \in M \times \mathcal{I}$ if $\Delta(m, \mathsf{i})$ is defined. We naturally extend these two functions to any sequence of inputs $u \in \mathcal{I}^+$, denoted $\mathsf{Post}^*_{\mathcal{M}}$ and $\mathsf{Out}^*_{\mathcal{M}}$. In particular, for $u \in \mathcal{I}^+$, $\mathsf{Post}^*_{\mathcal{M}}(m, u)$ is the state reached by $\mathcal{M}$ when reading u from m, while $\mathsf{Out}^*_{\mathcal{M}}(m, u)$ is the last output in $\mathcal{O}$ produced by $\mathcal{M}$ when reading u. The subcript $\mathcal{M}$ is ommitted when $\mathcal{M}$ is clear from the context. Now, the language $L(\mathcal{M})$ of finite words in $(\mathcal{I}\mathcal{O})^*$ accepted by $\mathcal{M}$ is defined as $L(\mathcal{M}) = \{\mathsf{i}_1\mathsf{o}_1 \dots \mathsf{i}_n\mathsf{o}_n \mid \forall 1 \leq j \leq n,\ \mathsf{Post}^*_{\mathcal{M}}(m_{\mathsf{init}}, \mathsf{i}_1 \dots \mathsf{i}_j) \text{ is defined and } \mathsf{o}_j = \mathsf{Out}^*_{\mathcal{M}}(m_{\mathsf{init}}, \mathsf{i}_1 \dots \mathsf{i}_j)\}$. The language $L_\omega(\mathcal{M})$ of infinite words accepted by $\mathcal{M}$ is the topological closure of $L(\mathcal{M})$: $L_\omega(\mathcal{M}) = \{w \in (\mathcal{I}\mathcal{O})^\omega \mid \mathsf{Prefs}(w) \cap (\mathcal{I}\mathcal{O})^* \subseteq L(\mathcal{M})\}$.

The reactive synthesis problem A *specification* is a language $\mathcal{S} \subseteq (\mathcal{I}\mathcal{O})^\omega$. The *reactive synthesis problem* (or just synthesis problem for short) is the problem of constructing, given a specification $\mathcal{S}$, a Mealy machine $\mathcal{M}$ such that $L_\omega(\mathcal{M}) \subseteq \mathcal{S}$ if it exists. Such a machine $\mathcal{M}$ is said to *realize* the specification $\mathcal{S}$, also written $\mathcal{M} \models \mathcal{S}$. We also say that $\mathcal{S}$ is *realizable* if some Mealy machine $\mathcal{M}$ realizes it. The induced decision problem is called the *realizability problem.*

It is well-known that if $\mathcal{S}$ is ω-regular (recognizable by, e.g., a parity automaton [33]) the realizability problem is decidable [1] and moreover, a Mealy machine realizing the specification can be effectively constructed. The realizability problem is 2ExpTime-Complete if $\mathcal{S}$ is given as an LTL formula [27] and ExpTime-Complete if $\mathcal{S}$ is given as a universal coBüchi automaton.

Theorem 1 ([7]). *The realizability problem for a specification $\mathcal{S}$ given as a universal coBüchi automaton $\mathcal{A}$ is* ExpTime-C. *Moreover, if $\mathcal{S}$ is realizable and $\mathcal{A}$ has n states, then $\mathcal{S}$ is realizable by a Mealy machine with $2^{O(n log_2 n)}$ states.*

We generalize this result to the following realizability problem which we describe first informally. Given a specification $\mathcal{S}$ and a preMealy machine $\mathcal{P}$, the goal is to decide whether $\mathcal{P}$ can be completed into a Mealy machine which realizes $\mathcal{S}$. We now define this problem formally. Given two preMealy machines $\mathcal{P}_1, \mathcal{P}_2$, we write $\mathcal{P}_1 \preceq \mathcal{P}_2$ if $\mathcal{P}_1$ is a subgraph of $\mathcal{P}_2$ in the following sense: there exists an injective mapping Φ from the states of $\mathcal{P}_1$ to the states of $\mathcal{P}_2$ which preserves the initial state (s_0 is the initial state of $\mathcal{P}_1$ iff $\Phi(s_0)$ is the initial state of $\mathcal{P}_2$) and the transitions ($\Delta_{\mathcal{P}_1}(p, \mathsf{i}) = (\mathsf{o}, q)$ iff $\Delta_{\mathcal{P}_2}(\Phi(p), \mathsf{i}) = (\mathsf{o}, \Phi(q))$. As a consequence, $L(\mathcal{P}_1) \subseteq L(\mathcal{P}_2)$ and $L_\omega(\mathcal{P}_1) \subseteq L_\omega(\mathcal{P}_2)$. Given a preMealy machine $\mathcal{P}$, we say that a specification $\mathcal{S}$ *is $\mathcal{P}$-realizable* if there exists a Mealy machine $\mathcal{M}$ such that $\mathcal{P} \preceq \mathcal{M}$ and $\mathcal{M}$ realizes $\mathcal{S}$. Note that if $\mathcal{P}$ is a (complete) Mealy machine, $\mathcal{S}$ is $\mathcal{P}$-realizable iff $\mathcal{P}$ realizes $\mathcal{S}$. The next result is proved in [5]:

Theorem 2. *Given a universal co-Büchi automaton $\mathcal{A}$ with n states defining a specification $\mathcal{S} = L^\forall(\mathcal{A})$ and a preMealy machine $\mathcal{P}$ with m states and n_h holes,* *deciding whether $\mathcal{S}$ is $\mathcal{P}$-realizable is* ExpTime*-hard and in* ExpTime *(in n and polynomial in m). Moreover, if $\mathcal{S}$ is $\mathcal{P}$-realizable, it is $\mathcal{P}$-realizable by a Mealy machine with $m + n_h 2^{O(n log_2 n)}$ states. Hardness holds even if $\mathcal{P}$ has two states and $\mathcal{A}$ is a deterministic reachability automaton.*

3 Synthesis from safety specifications and examples

In this section, we present the learning framework we use to synthesise Mealy machines from examples, and safety specifications. Its generalization to any ω-regular specification is described in Sec. 4 and solved by reduction to safety specifications. It is a two-phase algorithm: (1) it generalizes the examples while maintaining realizability of the specification, and outputs a preMealy machine, (2) it completes the preMealy machine into a full Mealy machine.

Phase 1: Generalizing the examples This phase exploits the examples by generalizing them as much as possible while maintaining realizability of the specification. It outputs a preMealy machine which is consistent with the examples and realizes the specification, if it exists. It is an RPNI-like learning algorithm [21,20] which includes specific tests to maintain realizability of the specification. In particular, it first builds a tree-shaped preMealy machine whose accepted language is exactly the set of prefixes $\mathsf{Prefs}(E)$ of the given set of examples E, called a *prefix-tree acceptor* (PTA). Then, it tries to merge as many as possible states of the PTA. The strategy used to select a state to merge another given state with, is a parameter of the algorithm, and is called a *merging strategy σ_G*. Formally, a *merging* strategy σ_G is defined over 4-tuples $(\mathcal{M}, m, E, X)$ where $\mathcal{M}$ is a preMealy machine, m is a state of $\mathcal{M}$, E is a set of examples and X is subset of states of $\mathcal{M}$ (the candidate states to merge m with), and returns a state of X, i.e., $\sigma_G(\mathcal{M}, m, E, X) \in X$.

The pseudo-code is given by alg. 1. Initially, it tests whether the set of examples E is consistent[1]and if yes, checks if $\mathsf{PTA}(E)$ can be completed into a Mealy machine realizing the given specification $\mathcal{S}$, thanks to Thm. 2. If that is the case, then it takes all prefixes of E as the set of examples, and enters a loop which consists in iteratively coarsening again and again some congruence $\sim$ over the states of $\mathsf{PTA}(E)$, by merging some of its classes. The congruence $\sim$ is initially the finest equivalence relation. It does the coarsening in a specific order: examples (which are states of $\mathsf{PTA}(E)$) are taken in length-lexicographic order. When entering the loop with example e, the algorithm computes at line 4 all the states, i.e., all the examples e' which have been processed already by the loop ($e' \prec_{ll} e$) and whose current class can be merged with the class of e (predicate $\mathsf{Mergeable}(\mathsf{PTA}(E), \sim, e, e')$). State merging is a standard operation in automata learning algorithms which intuitively means that merging the $\sim$-class of e and the $\sim$-class of e', and propagating this merge to the descendants of e and e', does not result any conflict. The formal definition is in [5]. At line 5, it filters the previous set by keeping only the states which, when merged with e, produce a preMealy

[1] E is consistent if outputs uniquely depends on prefixes. Formally, it means for all prefixes $u \in \mathsf{Prefs}(E) \cap (\mathcal{I}\mathcal{O})^*\mathcal{I}$, there is a unique output $\mathsf{o} \in \mathcal{O}$ s.t. $u\mathsf{o} \in \mathsf{Prefs}(E)$.

machine which can be completed into a Mealy machine realizing $\mathcal{S}$ (again by Thm. 2). If after the filtering there are still several candidates for merge, one of them is selected with the merging strategy σ_G and the equivalence relation is then coarsened via class merging (operation $\mathsf{MergeClass}(\mathsf{PTA}(E), \sim, e, e'))$. At the end, the algorithm returns the quotient of $\mathsf{PTA}(E)$ by the computed Mealy-congruence. As a side remark, when $\mathcal{S}$ is universal, i.e. $\mathcal{S} = (\mathcal{I}\mathcal{O})^\omega$, then it is realizable by *any* Mealy machine and therefore line 5 does not filter any of the candidates for merge. So, when $\mathcal{S}$ is universal, Algo 1 can be seen as an RPNI variant for learning preMealy machines.

Algorithm 1: GEN(E,$\mathcal{S}$,σ_G) – generalization algorithm

```
Input: A finite set of examples $E \subseteq (\mathcal{I}.\mathcal{O})^*$, a specification $\mathcal{S} \subseteq (\mathcal{I}.\mathcal{O})^\omega$ given
       as a deterministic safety automaton, a merging strategy $\sigma_G$
Output: A preMealy machine $\mathcal{M}$ s.t. $E \subseteq L(\mathcal{M})$ and $\mathcal{S}$ is $\mathcal{M}$-realizable, if it
        exists, otherwise UNREAL.
1 if $E$ is not consistent or $\mathcal{S}$ is not $\mathsf{PTA}(E)$-realizable then return UNREAL
2 $E \leftarrow \mathsf{Prefs}(E) \cap (\mathcal{I}\mathcal{O})^*$;   $\sim \leftarrow \{(e,e) \mid e \in E\}$;          // $\sim = diag_E$
3 for $e \in E$ in length-lexicographic order $\preceq_{ll}$ do
4     $mergeCand \leftarrow \{e' \mid \mathsf{Mergeable}(\mathsf{PTA}(E), \sim, e, e') \wedge e' \prec_{ll} e\}$
5     $mergeCand \leftarrow \{e' \in mergeCand \mid \mathcal{S}$ is $\mathsf{MergeStates}(\mathsf{PTA}(E), \sim$
        $, e, e')-realizable\}$
6     if $mergeCand \neq \varnothing$ then
7         $e' \leftarrow \sigma_G(\mathcal{M}, e, mergeCand)$
8         $\sim \leftarrow \mathsf{MergeClass}(\mathsf{PTA}(E), \sim, e, e')$
9 return $\mathsf{PTA}(E)/_\sim$
```

Phase 2: completion of preMealy machines into Mealy machines As it only constructs the PTA and tries to merge its states, the generalization phase might not return a (complete) Mealy machine. In other words, the machine it returns might still contain some holes (missing transitions). The objective of this second phase is to complete those holes into a Mealy machine, while realizing the specification. More precisely, when a transition is not defined from some state m and some input $\mathsf{i} \in \mathcal{I}$, the algorithm must select an output symbol $\mathsf{o} \in \mathcal{O}$ and a state m' to transition to, which can be either an existing state or a new state to be created (in that case, we write $m' = \mathsf{fresh}$ to denote the fact that m' is a fresh state). In our implementation, if it is possible to reuse a state m' that was created during the generalization phase, it is favoured over other states, in order to exploit the examples. However, the algorithm for the completion phase we describe now does not depend on any particular strategy to pick states. Therefore, it is parameterized by a *completion strategy* σ_C, defined over all triples $(\mathcal{M}, m, \mathsf{i}, X)$ where $\mathcal{M}$ is a preMealy machine with set of states M, (m, i) is a hole of $\mathcal{M}$, and $X \subseteq \mathcal{O} \times (M \cup \{\mathsf{fresh}\})$ is a list of candidate pairs (o, m'). It returns an element of X, i.e., $\sigma_C(\mathcal{M}, m, \mathsf{i}, X) \in X$.

In addition to σ_C, the completion algorithm takes as input a preMealy machine $\mathcal{M}_0$ and a specification $\mathcal{S}$, and outputs a Mealy machine which $\mathcal{M}_0$-realizes $\mathcal{S}$, if it exists. The pseudo-code is given in Algo 2. Initially, it tests whether $\mathcal{S}$

is $\mathcal{M}_0$-realizable, otherwise it returns UNREAL. Then, it keeps on completing holes of $\mathcal{M}_0$. The computation of the list of output/state candidates is done at the loop of line 5. Note that the **for**-loop iterates over $M \cup \{\mathsf{fresh}()\}$, where $\mathsf{fresh}()$ is a procedure that returns a fresh state not in M. The algorithm maintains the invariant that at any iteration of the **while**-loop, $\mathcal{S}$ is $\mathcal{M}$-realizable, thanks to the test at line 7, based on Thm. 2. Therefore, the list of candidates is necessarily non-empty. Amongst those candidates, a single one is selected and the transition on (m, i) is added to $\mathcal{M}$ accordingly at line 10.

Algorithm 2: COMP($\mathcal{M}_0$,$\mathcal{S}$,σ_C): preMealy machine completion algorithm

```
Input: A preMealy machine $\mathcal{M}_0 = (M, m_{\mathsf{init}}, \Delta)$, a specification $\mathcal{S} \subseteq (\mathcal{I}.\mathcal{O})^*$
       given as a deterministic safety automaton, a completion strategy $\sigma_C$
Output: A (complete) Mealy machine $\mathcal{M}$ such that $\mathcal{S}$ is $\mathcal{M}_0$-realizable,
        otherwise UNREAL.
1  if $\mathcal{S}$ is not $\mathcal{M}_0$-realizable then return UNREAL
2  $\mathcal{M} \leftarrow \mathcal{M}_0$
3  while there exists a hole $(m, \mathsf{i}) \in M \times \mathcal{I}$ do
4      candidates $\leftarrow \varnothing$
5      for $(\mathsf{o}, m') \in \mathcal{O} \times (M \cup \{fresh()\})$ do
                                        // fresh() denotes a new state not in $M$
6          $\mathcal{M}_{\mathsf{o},m'} \leftarrow (M \cup \{m'\}, m_{\mathsf{init}}, \Delta \cup \{(m, \mathsf{i}) \mapsto (\mathsf{o}, m')\})$
7          if $\mathcal{S}$ is $\mathcal{M}_{\mathsf{o},m'}$-realizable then
8              candidates $\leftarrow$ candidates $\cup \{(\mathsf{o}, m')\}$
9      $(\mathsf{o}, m') \leftarrow \sigma_C(\mathcal{M}, m, \mathsf{i}, candidates)$
10     $(M, \Delta) \leftarrow (M \cup \{m'\}, \Delta \cup \{(m, \mathsf{i}) \mapsto (\mathsf{o}, m')\})$
11     $\mathcal{M} \leftarrow (M, m_{\mathsf{init}}, \Delta)$
12 return $\mathcal{M}$
```

Two-phase synthesis algorithm from specifications and examples The two-phase synthesis algorithm for safety specifications and examples, called SYNTHSAFE($E, \mathcal{S}, \sigma_G, \sigma_C$) works as follows: it takes as input a set of examples E, a specification $\mathcal{S}$ given as a deterministic safety automaton, a generalizing and completion strategies σ_G, σ_C respectively. It returns a Mealy machine $\mathcal{M}$ which realizes $\mathcal{S}$ and such that $E \subseteq L(\mathcal{M})$ if it exists. In a first steps, it calls GEN($E, \mathcal{S}, \sigma_G$). If this calls returns UNREAL, then SYNTHSAFE return UNREAL as well. Otherwise, the call to GEN returns a preMealy machine $\mathcal{M}_0$. In a second step, SYNTHSAFE calls COMP($\mathcal{M}_0, \mathcal{S}, \sigma_C$). If this call returns UNREAL, so does SYNTHSAFE, otherwise SYNTHSAFE returns the Mealy machine computed by COMP. The pseudo-code of SYNTHSAFE can be found in [5].

The completion procedure may not terminate for some completion strategies. It is because the completion strategy could for instance keep on selecting pairs of the form (o, m') where m' is a fresh state. However we prove that it always terminates for *lazy* completion strategies. A completion strategy σ_C is said to be *lazy* if it favours existing states, which formally means that if $X \setminus (\mathcal{O} \times \{\mathsf{fresh}\}) \neq \varnothing$, then $\sigma_C(\mathcal{M}, m, \mathsf{i}, X) \notin \mathcal{O} \times \{\mathsf{fresh}\}$. The 1st theorem states correctness and termination of the algorithm for lazy completion strategies (assuming the functions σ_G and σ_C are computable in worst-case exptime in the size of their inputs).

Theorem 3 (termination and correctness)**.** *For all finite sets of examples $E \subseteq (\mathcal{I}.\mathcal{O})^*$, all specifications $\mathcal{S} \subseteq (\mathcal{I}.\mathcal{O})^\omega$ given as a deterministic safety automaton $\mathcal{A}$ with n states, all merging strategies σ_G and all completion strategies σ_C, if* SynthSafe*$(E, \mathcal{S}, \sigma_G, \sigma_C)$ terminates then, it returns a Mealy machine $\mathcal{M}$ such that $E \subseteq L(\mathcal{M})$ and $\mathcal{M}$ realizes $\mathcal{S}$, if it exists, otherwise it returns UNREAL. Moreover,* SynthSafe*$(E, \mathcal{S}, \sigma_G, \sigma_C)$ terminates if σ_C is lazy, in worst-case exponential time (polynomial in the size*[2] *of E and exponential in n).*

The proof of the latter theorem is a consequence of several results proved on the generalization and completion phases, and is given in [5].

A Mealy machine $\mathcal{T}$ is minimal if for all Mealy machine $\mathcal{M}$ such that $L(\mathcal{T}) = L(\mathcal{M})$, the number of states of $\mathcal{M}$ is at least that of $\mathcal{T}$. The next result, proved in [5], states that any minimal Mealy machine realizing a specification $\mathcal{S}$ can be returned by our synthesis algorithm, providing representative examples.

Theorem 4 (Mealy completeness)**.** *For all specifications $\mathcal{S} \subseteq (\mathcal{I}.\mathcal{O})^\omega$ given as a deterministic safety automaton, for all minimal Mealy machines $\mathcal{M}$ realizing $\mathcal{S}$, there exists a finite set of examples $E \subseteq (\mathcal{I}.\mathcal{O})^*$, of size polynomial in the size of $\mathcal{M}$, such that for all generalizing strategies σ_G and completion strategies σ_C, and all sets of examples E' s.t. $E \subseteq E' \subseteq L(\mathcal{M})$,* SynthSafe*$(E', \mathcal{S}, \sigma_G, \sigma_C) = \mathcal{M}$.*

The polynomial upper bound given in the statement of Theorem 4 is more precisely the following: the cardinality of E is $O(m + n^2)$ where n is the number of states of $\mathcal{M}$ while m is its number of transitions. Moreover, each example $e \in E$ has length $O(n^2)$. More details can be found in Remark 1 of [5].

4 Synthesis from ω-regular specifications and examples

We now consider the case where the specification $\mathcal{S}$ is given as universal coBüchi automaton, in Section 4. We consider this class of specifications as it is complete for ω-regular languages and allow for compact symbolic representations. Further in this section, we consider the case of LTL specifications.

Specifications given as universal coBüchi automata Our solution for ω-regular specifications relies on a reduction to the safety case treated in Sec. 3. It relies on previous works that develop so called Safraless algorithms for ω-regular reactive synthesis [24,29,18]. The main idea is to strengthen the (safety) acceptance condition of the automaton from coBüchi to K-coBüchi. It is complete for the plain synthesis problem (w/o examples) if K is large enough (in the worst-case exponential in the number of states of the automaton (e.g., see [18])). Moreover, it allows for incremental synthesis algorithms: if the specification defined by the automaton with a k-coBüchi acceptance condition is realizable, for $k \leq K$, so is the specification defined by taking K-coBüchi acceptance. Here, as we also take examples into account, we need to slightly adapt the results. The next theorem is proved in [5] while the next lemma is immediate:

[2] The size of E is the sum of the lengths of the examples of E.

Theorem 5. *Given a universal co-Büchi automaton $\mathcal{A}$ with n states defining a specification $\mathcal{S} = L^{\forall}(\mathcal{A})$ and a preMealy machine $\mathcal{P}$ with m states, we have that $\mathcal{S}$ is $\mathcal{P}$-realizable iff $\mathcal{S}' = L^{\forall}_{K}(A)$ is $\mathcal{P}$-realizable for $K = nm|\mathcal{I}|2^{\mathbf{O}(n \log_2 n)}$.*

Lemma 1. *For all co-Büchi automata $\mathcal{A}$, for all preMealy machines $\mathcal{P}$, for all $k_1 \leq k_2$, we have that $L^{\forall}_{k_1}(\mathcal{A}) \subseteq L^{\forall}_{k_2}(\mathcal{A})$ and so if $L^{\forall}_{k_1}(\mathcal{A})$ is $\mathcal{P}$-realizable then $L^{\forall}_{k_2}(\mathcal{A})$ is $\mathcal{P}$-realizable. Furthermore for all $k \geq 0$, if $\mathcal{S}' = L^{\forall}_{k}(A)$ is $\mathcal{P}$-realizable then $\mathcal{S} = L^{\forall}(\mathcal{A})$ is $\mathcal{P}$-realizable.*

Thanks to the latter two results applied to $\mathcal{P} = \mathsf{PTA}(E)$ for a set E of examples of size m, we can design an algorithm for synthesising Mealy machines from a specification defined by a universal coBüchi automaton $\mathcal{A}$ with n states and E: it calls SynthSafe on the safety specification $L^{\forall}_{k}(\mathcal{A})$ and E for increasing values of k, until it concludes positively, or reach the bound $K = 2^{\mathbf{O}(mn \log_2 mn)} + 1$. In the latter case, it returns UNREAL. However, to apply SynthSafe properly, $L^{\forall}_{k}(\mathcal{A})$ must be represented by a deterministic safety automaton. This is possible as k-coBüchi automata are determinizable [18].

Determinization The determinization of k-co-Büchi automata $\mathcal{A}$ relies on a simple generalization of the subset construction: in addition to remembering the set of states that can be reached by a prefix of a run while reading an infinite word, the construction counts the maximal number of times a run prefix that reaches a given state q has visited states labelled with color 1 (remember that a run can visit at most k such states to be accepting). The states of the deterministic automaton are so-called *counting functions*, formally defined for a co-Büchi automaton $\mathcal{A} = (Q, q_{\mathsf{init}}, \Sigma, \delta, d)$ and $k \in \mathbb{N}$, as the set noted $CF(\mathcal{A}, k)$ of functions $f : Q \to \{-1, 0, 1, \dots, k, k+1\}$. If $f(q) = -1$ for some state q, it means that q is inactive (no run of $\mathcal{A}$ reach q on the current prefix). The initial counting function f_{init} maps all 1-colored initial states to 1, all 0-colored initial states to 0 and all other states to -1. We denote by $\mathcal{D}(\mathcal{A}, k) = (Q^{\mathcal{D}} = CF(\mathcal{A}, k), q^{\mathcal{D}}_{\mathsf{init}} = f_{\mathsf{init}}, \Sigma, \delta^{\mathcal{D}}, Q^{\mathcal{D}}_{\mathsf{usf}})$ the deterministic automaton obtained by this determinization procedure. It is formally defined in [5]. We can now give algorithm SynthLearn, in pseudo-code, as Algo 3.

Complexity considerations and improving the upper-bound As the automaton $\mathcal{D}(\mathcal{A}, k)$ is in the worst-case exponential in the size of the automaton $\mathcal{A}$, a direct application of Thm. 3 yields a doubly exponential time procedure. This complexity is a consequence of the fact that the $\mathcal{P}$-realizability problem is ExpTime in the size of the deterministic automaton as shown in Thm. 2, and that the termination of the completion procedure is also worst-case exponential in the size of the deterministic automaton.

We show that we can improve the complexity of each call to SynthSafe and obtain an optimal worst-case (single) exponential complexity. We provide an algorithm to check $\mathcal{P}$-realizability of a specification $\mathcal{S} = L^{\forall}_{k}(\mathcal{A})$ that runs in time singly exponential in the size of $\mathcal{A}$ and polynomial in k and the size of $\mathcal{P}$. Second, we provide a finer complexity analysis for the termination of the completion algorithm, which exhibits a worst case exponential time in $|\mathcal{A}|$. Those two improvements lead to an overall complexity of SynthLearn which is expo-

Algorithm 3: SynthLearn(E,$\mathcal{A}$,σ_G,σ_C) – synthesis algorithm from ω-regular specification and examples by a reduction to safety

```
Input: A universal co-Büchi automaton $\mathcal{A}$ with $n$ states, a finite set of
       examples $E \subseteq (\mathcal{I}.\mathcal{O})^*$, a generalizing strategy $\sigma_G$ and a completion
       strategy $\sigma_C$.
Output: A Mealy machine $\mathcal{M}$ realizing $L^{\forall}(\mathcal{A})$ and such that $E \subseteq L(\mathcal{M})$ if it
        exists, otherwise UNREAL.
1 $K \leftarrow nm|\mathcal{I}|2^{\mathbf{O}(n \log_2 n)}$; $k \leftarrow 0$;          // $m$ is the size of $E$
2 while $k \leq K$ do
3     if SynthSafe($E, \mathcal{D}(\mathcal{A},k), \sigma_C, \sigma_G$) $\neq$ $UNREAL$ then
4         return SynthSafe($E, \mathcal{D}(\mathcal{A},k), \sigma_C, \sigma_G$)
5     $k \leftarrow k+1$;
6 return $UNREAL$
```

nential in the size of the specification $\mathcal{A}$ and polynomial in the set of examples E. This is provably worst-case optimal because for $E = \emptyset$ the problem is already ExpTime-Complete. We explain next the first improvement, the upper-bound for termination is provided in [5].

Checking $\mathcal{P}$-realizability of a specification $\mathcal{S} = L_k^{\forall}(\mathcal{A})$ To obtain a better complexity, we exploit some structure that exists in the deterministic automaton $\mathcal{D}(\mathcal{A},k)$. First, the set of counting functions $CF(\mathcal{A},k)$ forms a complete lattice for the partial order $\preceq$ defined by $f_1 \preceq f_2$ if $f_1(q) \leq f_2(q)$ for all states q. We denote by $f_1 \bigsqcup f_2$ the least upper-bound of f_1, f_2, and by $W_k^{\mathcal{A}}$ the set of counting functions f such that the specification $L(\mathcal{D}(\mathcal{A},k)[f])$ is realizable (i.e. the specification defined by $\mathcal{D}(\mathcal{A},k)$ with initial state f). It is known that $W_k^{\mathcal{A}}$ is downward-closed for $\preceq$ [18], because for all $f_1 \preceq f_2$, any machine realizing $L(\mathcal{D}(\mathcal{A},k)[f_2])$ also realizes $L(\mathcal{D}(\mathcal{A},k)[f_1])$. Therefore, $W_k^{\mathcal{A}}$ can be represented compactly by the antichain $\lceil W_k^{\mathcal{A}} \rceil$ of its $\preceq$-maximal elements. Now, the first improvement is obtained thanks to the following result:

Lemma 2. *Given a preMealy $\mathcal{P} = (M, m_0, \Delta)$, a co-Büchi automata $\mathcal{A}$, and $k \in \mathbb{N}$. For all states $m \in M$, we let $F^*(m) = \bigsqcup\{f \mid \exists u \in (\mathcal{I}\mathcal{O})^* \cdot \mathsf{Post}^*_{\mathcal{P}}(m_0,u) = m \wedge \mathsf{Post}_{\mathcal{D}}(f_0,u) = f\}$. Then, $L(\mathcal{D}(\mathcal{A},k))$ is $\mathcal{P}$-realizable iff there does not exist $m \in M$ such that $F^*(m) \notin W_k^{\mathcal{A}}$.*

It is easily shown that the operator F^* can be computed in pTime. Thus, the latter lemma implies that there is a poly-time algorithm in $|\mathcal{P}|, |\mathcal{A}|, k \in \mathbb{N}$, and the size of $\lceil W_k^{\mathcal{A}} \rceil$ to check the $\mathcal{P}$-realizability of $L^{\forall}(\mathcal{A})$. Formal details in [5].

We end this subsection by summarizing the behavior of our synthesis algorithm for ω-regular specifications defined as universal co-Büchi automata.

Theorem 6. *Given a universal coBüchi automaton $\mathcal{A}$ and a set of examples E, the synthesis algorithm* SynthLearn *returns, if it exists, a Mealy machine $\mathcal{M}$ such that $E \subseteq L(\mathcal{M})$ and $L_\omega(\mathcal{M}) \subseteq L^{\forall}(\mathcal{A})$, in worst-case exponential time in the size of $\mathcal{A}$ and polynomial in the size of E. Otherwise, it returns UNREAL.*

Specifications given as an LTL formula We are now in position to apply Alg. 3 to a specification given as LTL formula φ. Indeed, thanks to the results of the subsection above, to provide an algorithm for LTL specifications, we only need to translate φ into a universal co-Büchi automaton. This can be done according to the next lemma. It is well-known (see [24]), that given an LTL formula φ over two sets of atomic propositions $P_{\mathcal{I}}$ and $P_{\mathcal{O}}$, we can construct in exponential time a universal co-Büchi automaton $\mathcal{A}_\varphi$ such that $L^\forall(\mathcal{A}_\varphi) = [\![\varphi]\!]$, i.e. $\mathcal{A}$ recognizes exactly the set of words $w \in (2^{P_{\mathcal{I}}} 2^{P_{\mathcal{O}}})^\omega$ that satisfy φ. We then get the following theorem that gives the complexity of our synthesis algorithm for a set of examples E and an LTL formula φ, complexity which is provably worst-case optimal as deciding if $[\![\varphi]\!]$ is realizable with $E = \emptyset$, i.e. the plain LTL realizability problem, is already 2ExpTime-Complete [27].

Theorem 7. *Given an LTL formula φ and a set of examples E, the synthesis algorithm* SynthLearn *returns a Mealy machine $\mathcal{M}$ such that $E \subseteq L(\mathcal{M})$ and $L_\omega(\mathcal{M}) \subseteq [\![\varphi]\!]$ if it exists, in worst-case doubly exponential time in the size of φ and polynomial in the size of E. Otherwise it returns UNREAL.*

5 Implementation and Case study

We have implemented the algorithm SynthLearn of the previous section in a prototype tool, in Python, using the tool Acacia-Bonzai [11] to manipulate antichains of counting functions. We first explain the heuristics we have used to define state-merging and completion strategies, and then demonstrate how our implementation behaves on a case study whose goal is to synthesize the controller for an elevator. The interested reader can find in [5] other case studies, including a controller for an e-bike and two variations on mutual exclusion.

Merging and completion strategies implemented in our prototype Our tool implements a *merging* strategy σ_G where, given an example e that leads in the current preMealy machine to a state m and a set $\{m_1, m_2, \dots, m_k\}$ of candidates for merging, as computed in line 7 of Algorithm 1, we choose state m_i with a $\preceq$-minimal counting function $F^*(m_i)$, as defined in Lemma 2. Intuitively, favouring minimal counting functions preserves as much as possible the set of behaviors that are possible after the example e.

Our tool also implements a *completion strategy* σ_C, where for every hole (m, i) of the preMealy machine $\mathcal{M}$ and out of the list of candidate pairs, selects an element which again favour states associated with $\preceq$-minimal counting functions. For more details, we refer the reader to [5].

Lift Controller Example We illustrate how to use our tool to construct a suitable controller for a two-floor elevator system.

Considering two floors is sufficient enough to illustrate most of the main difficulties of a more general elevator. Inputs of the controller are given by two atomic propositions `b0` and `b1`, which are true whenever the button at floor 0 (resp. floor 1) is pressed by a user. Outputs are given by the atomic propositions `f0` and `f1`, true whenever the elevator is at floor 0 (resp. floor 1); and `ser`, true whenever the elevator is *serving* the current floor (i.e. doors are opened). This controller should ensure the following core properties:

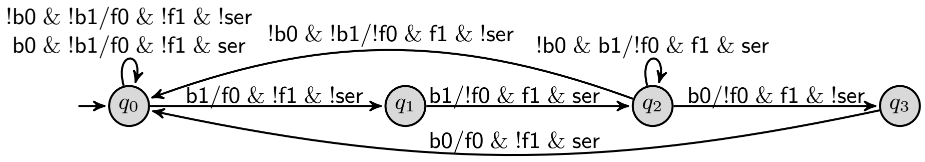

Fig. 2: Machine returned by our tool on the elevator specification w/o examples. Here, q_0, q_1, q_2, q_3 represents the states where `f0` is served when required, where `b1` is pending, where `f1` is served, the state where `b0` is pending respectively.

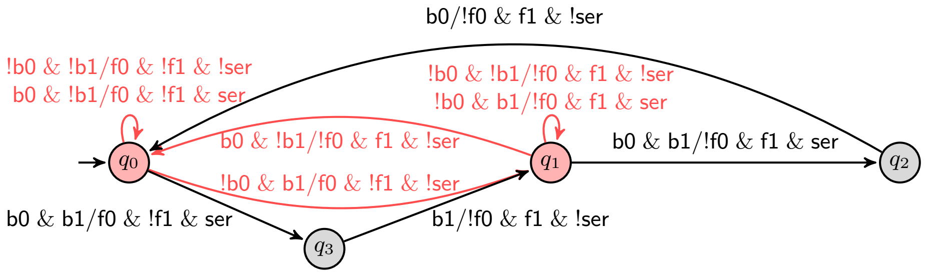

Fig. 3: Mealy machine returned by our tool on the elevator specification with additional examples. The preMealy machine obtained after generalizing the examples and before completion is highlighted in red. This took 3.10s to be generated.

1. **Functional Guarantee:** whenever a button of floor 0 (resp. floor 1) is pressed, the elevator must eventually *serve* floor 0 (resp. floor 1): `G(b0 -> F (f0 & ser)) & G(b1 -> F (f1 & ser))`
2. **Safety Guarantee:** The elevator is always at one floor exactly: `G(f0<->!f1)`
3. **Safety Guarantee:** The elevator cannot transition between two floors when doors are opened: `G((f0 & ser) -> X(!f1)) & G((f1 & ser) -> X(!f0))`
4. **Initial State:** The elevator should be in floor 0 initially: `f0`

Additionally, we make the following **assumption**: whenever a button of floor 0 (or floor 1) is pressed, it must remain pressed until the floor has been served, i.e., `G(b0 -> (b0 W (f0 & ser))) & G(b1 -> (b1 W (f1 & ser)))`.

Before going into the details of this example, let us explain the methodology that we apply to use our tool on this example. We start by providing only the high level specification φ_{CORE} for the elevator given above. We obtain a first Mealy machine from the tool. We then observe the machine to identify prefix of behaviours that we are unhappy with, and for which we can provide better alternative decisions. Then we run the tool on φ_{CORE} and the examples that we have identified, and we get a new machine, and we proceed like that up to a point where we are satisfied with the synthesized Mealy machine.

Let us now give details. When our tool is provided with this specification without any examples, we get the machine depicted in fig. 2. This solution makes the controller switch between floor 0 and floor 1, sometimes unnecessarily. For instance, consider the trace `s # {!b0 & !b1}{!f0 & f1 & !ser} # {!b0 & !b1}{f0 & !f1 & !ser}`, where we let `s = {!b0 & b1}{f0 & !f1 & !ser}`

`# {!b0 & b1}{!f0 & f1 & ser}`. Here, we note that the transition goes back to state q_0, where the elevator is at floor 0, when the elevator could have remained at floor 1 after serving floor 1. The methodology described above allows us to identify the following three examples:

1. The 1st trace states that after serving floor 1, the elevator must remain at floor 1 as `b0` is false: `s # {!b0 & !b1}{!f0 & f1 & !ser} # {!b0 & !b1}{!f0 & f1 & !ser}`
2. The 2nd trace states that the elevator must remain at floor 0, as `b1` is false: `{!b0 & !b1}{f0 & !f1 & !ser} # {!b0 & !b1}{f0 & !f1 & !ser}`
3. The 3rd trace ensures that after `s`, there is no unnecessary delay in serving floor 0 after floor 1 is served in `s`: `s # {b0 & !b1}{!f0 & f1 & !ser} # {b0 & !b1}{f0 & !f1 & ser}`

With those additional examples, our tool outputs the machine of fig. 3, which generalizes them and now ensures that moves of the elevator occur only when required. For example, the end of the first trace has been generalized into a loop on state q_1 ensuring that the elevator does not go to floor 0 from floor 1 unless `b0` is pressed. We note that the number of examples provided here is much smaller than the theoretical (polynomial) upper bound proved in Theorem 4.

6 Conclusion

We have introduced *synthesis with a few hints*, which allows the user to guide synthesis using examples of expected executions of high quality solutions. Existing synthesis tools may provide unnatural solutions when fed with high-level specifications only. As providing complete specifications goes against the very goal of synthesis, we believe our algorithm has a greater potential in practice.

We have studied the computational complexity of problems that need to be solved during our synthesis procedure. We have proved our algorithm is *complete*: any Mealy machine $\mathcal{M}$ realizing a specification φ can be obtained from φ and a representative example set E, whose size is bounded polynomially in the size of $\mathcal{M}$. We have implemented our algorithm in a prototype tool that extends Acacia-Bonzai [11] with tailored state-merging learning algorithms. We have shown that only a small number of examples are necessary to obtain high quality machines from high-level LTL specifications only. The tool is not fully optimized yet. While this is sufficient to demonstrate the relevance of our approach, we will work on efficiency aspects of the implementation.

As future works, we will consider extensions of the user interface to interactively and concisely specify sets of (counter-)examples to solutions output by the tool. In the same line, an interesting future direction is to handle parametric examples (e.g. elevator with the number of floors given as parameter). This would require to provide a concise syntax to define parametric examples and to design efficient synthesis algorithm in this setting. We will also consider the possibility to formulate negative examples, as our theoretical results readily extend to this case and their integration in the implementation should be easy.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings. Lecture Notes in Computer Science, vol. 372, pp. 1–17. Springer (1989)
2. Almagor, S., Kupferman, O., Velner, Y.: Minimizing expected cost under hard boolean constraints, with applications to quantitative synthesis. In: 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 9:1–9:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
3. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, pp. 1–25 (2015)
4. Alur, R., Martin, M.M.K., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8855, pp. 75–91. Springer (2014)
5. Balachander, M., Filiot, E., Raskin, J.F.: Ltl reactive synthesis with a few hints (2023). https://doi.org/10.48550/ARXIV.2301.10485, https://arxiv.org/abs/2301.10485
6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 140–156. Springer (2009)
7. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Handbook of Model Checking, pp. 921–962. Springer (2018)
8. Bloem, R., Chockler, H., Ebrahimi, M., Strichman, O.: Synthesizing non-vacuous systems. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 55–72. Springer International Publishing, Cham (2017)
9. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 652–657. Springer (2012)
10. Bruyère, V., Filiot, E., Randour, M., Raskin, J.: Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. Inf. Comput. **254**, 259–295 (2017). https://doi.org/10.1016/j.ic.2016.10.011, https://doi.org/10.1016/j.ic.2016.10.011
11. Cadilhac, M., Pérez, G.A.: Acacia-bonsai: A modern implementation of downset-based LTL realizability. CoRR **abs/2204.06079** (2022). https://doi.org/10.48550/arXiv.2204.06079, https://doi.org/10.48550/arXiv.2204.06079
12. Damas, C., Lambeau, B., van Lamsweerde, A.: Scenarios, goals, and state machines: a win-win partnership for model synthesis. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006. pp. 197–207. ACM (2006)

13. D'Ippolito, N., Braberman, V.A., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. ACM Trans. Softw. Eng. Methodol. **22**(1), 9:1–9:36 (2013). https://doi.org/10.1145/2430536.2430543, https://doi.org/10.1145/2430536.2430543
14. Dupont, P., Lambeau, B., Damas, C., van Lamsweerde, A.: The QSM algorithm and its application to software behavior model induction. Appl. Artif. Intell. **22**(1&2), 77–115 (2008). https://doi.org/10.1080/08839510701853200, https://doi.org/10.1080/08839510701853200
15. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Gbaguidi, A., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From spot 2.0 to spot 2.10: What's new? CoRR **abs/2206.11366** (2022). https://doi.org/10.48550/arXiv.2206.11366, https://doi.org/10.48550/arXiv.2206.11366
16. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 426–442 (2017)
17. Faymonville, P., Finkbeiner, B., Tentrup, L.: Bosy: An experimentation framework for bounded synthesis. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 325–332. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_17, https://doi.org/10.1007/978-3-319-63390-9_17
18. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 263–277. Springer (2009)
19. Filiot, E., Jin, N., Raskin, J.: Antichains and compositional algorithms for LTL synthesis. Formal Methods Syst. Des. **39**(3), 261–296 (2011). https://doi.org/10.1007/s10703-011-0115-3, https://doi.org/10.1007/s10703-011-0115-3
20. Giantamidis, G., Tripakis, S., Basagiannis, S.: Learning Moore machines from input-output traces. Int. J. Softw. Tools Technol. Transf. **23**(1), 1–29 (2021)
21. Heinz, J., de la Higuera, C., van Zaanen, M.: Grammatical Inference for Computational Linguistics. Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers (2015). https://doi.org/10.2200/S00643ED1V01Y201504HLT028, https://doi.org/10.2200/S00643ED1V01Y201504HLT028
22. Kupferman, O.: On high-quality synthesis. In: Computer Science - Theory and Applications - 11th International Computer Science Symposium in Russia, CSR 2016, St. Petersburg, Russia, June 9-13, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9691, pp. 1–15. Springer (2016)
23. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods. pp. 82–98. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
24. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings. pp. 531–542. IEEE Computer Society (2005)

25. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
26. Natarajan, N., Simmons, D., Datha, N., Jain, P., Gulwani, S.: Learning natural programs from a few examples in real-time. In: Chaudhuri, K., Sugiyama, M. (eds.) The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan. Proceedings of Machine Learning Research, vol. 89, pp. 1714–1722. PMLR (2019), http://proceedings.mlr.press/v89/natarajan19a.html
27. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings. Lecture Notes in Computer Science, vol. 372, pp. 652–671. Springer (1989)
28. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning fragments of linear temporal logic. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 263–280. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_14, https://doi.org/10.1007/978-3-030-99524-9_14
29. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4762, pp. 474–488. Springer (2007)
30. Singh, R., Gulwani, S.: Transforming spreadsheet data types using examples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 343–356 (2016). https://doi.org/10.1145/2837614.2837668, https://doi.org/10.1145/2837614.2837668
31. Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013), https://doi.org/10.1007/s10009-012-0249-7
32. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907, https://doi.org/10.1145/1168857.1168907
33. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (1991)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.