Synthesis of Compact Strategies for Coordination Programs

Kedar S. Namjoshi[1] (✉) and Nisarg Patel[2]

[1] Nokia Bell Labs, Murray Hill, NJ 07974, USA
kedar.namjoshi@nokia-bell-labs.com
[2] New York University, New York, NY 10001, USA nisarg@nyu.edu

Abstract. In multi-agent settings, such as IoT and robotics, it is necessary to coordinate the actions of independent agents to achieve a joint behavior. While it is often easy to specify the desired behavior, programming the necessary coordination can be difficult. This makes coordination an attractive target for automated program synthesis; however, current methods may produce strategies that issue useless actions. This paper develops theory and methods to synthesize coordination strategies that are guaranteed not to initiate unnecessary actions. We refer to such strategies as being "compact." We formalize the intuitive notion of compactness, show that existing methods do not guarantee compactness, and propose a solution. The solution transforms a given temporal logic specification using automata-theoretic constructions to incorporate a notion of minimality. The central result is that the winning strategies for the transformed specification are precisely the compact strategies for the original. One can therefore apply known synthesis methods to produce compact strategies. We report on prototype implementations that synthesize compact strategies for temporal logic specifications and for specifications of multi-robot coordination.

1 Introduction

Imagine a future home where devices are network-controllable and the control program is synthesized from requirements. Suppose that the homeowner asks for the living-room lights to be turned on when it gets dark. To meet this requirement, a control program must necessarily coordinate the on/off state of the lights with readings from an illumination sensor.

This specification may be expressed more precisely in linear-time temporal logic (LTL) as $\mathsf{G}(\text{dark} \Rightarrow \mathsf{X}\,\text{light-on})$.[3] Here "dark" is a proposition that represents a reading from the sensor, and is therefore an input to the control program, while "light-on" is a proposition that represents an action, and is therefore an output of the control program. Abstracting this formula to the shape $\mathsf{G}(a \Rightarrow \mathsf{X}\,b)$, the left half of Figure 1 shows the smallest state machine that

[3] G and X are, respectively, the temporal always and next-time operators. Actions are assumed instantaneous for simplicity.

© The Author(s) 2022
D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 46–63, 2022.
https://doi.org/10.1007/978-3-030-99524-9_3

meets this specification. It represents a control program that entirely ignores the sensor input and leaves the lights on all day! This strategy is clearly undesirable, although technically it does meet the specification. The machine on the right represents the "commonsense" controller that keeps the lights on only as long as the sensor indicates that it is dark. The two machines are equally valid from the viewpoint of correctness. How then should we distinguish them? And how can a synthesis method avoid generating undesirable solutions? Those are the questions addressed in this paper.

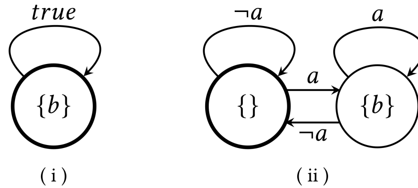

Fig. 1. Non-compact (left) and compact (right) machines for $\mathsf{G}(a \Rightarrow \mathsf{X}\, b)$. The initial state is indicated by a thick border. Output actions are listed at each state; input conditions are placed on the edges.

We suggest that the crucial distinguishing factor is that the left-hand machine invokes actions that are not essential to satisfying the property. For instance, if the input a is false now, there is no need to invoke action b in the next step. If input a remains false, there is no need to invoke action b at all. It is vital to avoid useless actions in the domains of IoT and robotics, where agents interact with the physical world: there is no need to switch on a toaster when only watering the lawn is asked for. Indeed, switching on the toaster unexpectedly may have dangerous side effects. A reader may easily imagine other similar situations.

We refer to the policy of avoiding unnecessary actions as *compactness*. Strategies that satisfy this property while meeting the specification are called *compact*. An immediate question is whether compactness is ensured by standard synthesis methods. Unfortunately, the answer is 'no.' Bounded synthesis [35,20], for instance, will produce the smallest satisfying Mealy or Moore machine; in this setting, the solution of Figure 1(i). We have validated this experimentally with the tool BoSy [19]. Quantitative synthesis (cf. [6]) finds solutions that are worst-case optimal, i.e., programs where the maximum cost, over all input sequences, is the lowest possible. (Dually, programs where the worst-case reward is the highest possible.) Letting each action invocation have unit cost, a quantitative method cannot distinguish between the solutions shown, as both have the same maximum cost for the input where a is always true. We make this analysis precise subsequently, and show that average-case optimality also does not always distinguish compact from non-compact solutions. We have validated this experimentally with the tool QUASY [13]. Hence, compactness cannot be defined in quantitative terms: the synthesis of compact strategies requires new methods.

At its core, the issue of compactness is a variation of the well-known *frame problem* in logic-based AI [29]. The natural way to express the example requirement is as $\mathsf{G}(a \Rightarrow \mathsf{X}\, b)$. However, the semantics of temporal logic allows many satisfying interpretations; among those is the undesirable one of Figure 1(i). This tension between the freedom of interpretation allowed in logic and the naturalness of a specification is at the heart of the frame problem. One approach to achieving compactness is therefore to write a tighter specification, which permits fewer interpretations; e.g., to write the stronger assertion $\mathsf{G}(a \equiv \mathsf{X}\, b)$. But this is not a natural choice. Moreover, reworking a specification by hand to rule out interpretations with unnecessary actions is difficult as the process is not compositional: i.e., one cannot rework portions of a specification separately. The specification transformation defined here performs such a tightening automatically, using automata-theoretic constructions.

The motivating application of compactness is to the synthesis of centralized coordination programs. As formalized in [3], in a coordination problem, a group of independent agents, denoted $A_1, \ldots, A_n$, are guided by an additional synthesized agent, C, so that their joint behavior meets a temporal specification φ. That work describes a specification transformation from φ to φ' that incorporates asynchronous agent behavior and other constraints. This transformation, however, does not guarantee compactness. We take the transformed problem as the starting point for our investigations, and consider the more general question of how to generate a compact solution for a given temporal specification.

We begin by proposing a mathematical definition of the compactness property. Generalizing from the example, one can consider a strategy to be compact if for each input sequence, the sequence of actions produced as output (1) meets the specification and (2) cannot be further improved. We formalize the second notion as minimality with respect to a supplied "better than" preference relation between two output sequences. This formulation is closely related to formalizations of commonsense reasoning, in particular the notion of circumscription introduced by McCarthy in [28].

For coordination problems, a natural preference relation is based on the subset ordering on sets of actions. We say that sequence y is better than sequence x if (1) in each step, the actions issued in y are a subset of the actions issued by x and (2) for at least one step, the actions in y are a strict subset of the actions in x. The smallest compact strategy for $\mathsf{G}(a \Rightarrow \mathsf{X}\, b)$ under this preference relation issues action b precisely when input a is true at the prior step. Otherwise, there is a point where a is false but b is issued at the next step. Removing this occurrence of b produces a better sequence that also satisfies the property. This is precisely the strategy defined by the machine in Fig. 1(ii). Alternative preference relations may order sets of actions by size, or order sequences of actions by the substring relation. One may also compare infinite action sequences by cost (limit average or discounted sum) using comparator automata [2]. The choice of preference relation is driven by the application domain. To accommodate various options, compactness is parameterized by the preference relation.

Technically, a temporal specification φ can be viewed as a language L, a set of infinite words over a joint input-output alphabet. For a preference relation $\prec$ over infinite words, it is natural to formulate the language $\min(L, \prec)$ that contains only the *minimal words* in L with respect to the preference relation. The central theoretical result in this paper is that there is a compact strategy satisfying L if, and only if, there is a strategy satisfying $\min(L, \prec)$. This theorem reduces the question of synthesizing compact strategies to a standard synthesis question, making it possible to use existing synthesis algorithms to produce compact strategies. We give sufficient conditions under which $\min(L, \prec)$ is regular when L is a regular language, and show how to effectively construct a finite automaton for the minimal language and for its complement, from either an automaton or an LTL formula for L. The constructed automata can also be used to model-check whether a given control program defines a compact strategy. Moreover, the transformation makes it possible to modularly apply quantitative or other criteria for synthesis from $\min(L, \prec)$; for instance, to synthesize compact strategies that minimize program size or worst-case execution time.

We have implemented these constructions and used them to synthesize compact strategies for LTL specifications and for a class of specifications that arise in multi-robot coordination. Experiments show that compact strategies exist for many specifications and can be effectively computed, albeit with some added overhead. We also experiment with approximation methods which are simpler and avoid potential worst-case exponential blowups in the general construction.

In our view, the main contributions of this work are in bringing attention to the need for compactness in program synthesis; showing its independence from existing criteria; giving a precise formulation in terms of minimality; and in designing and implementing algorithms to synthesize compact strategies.

2 Background

Automata A finite automaton is a tuple $(Q, \Sigma, \hat{Q}, \delta, F)$ where Q is a set of *states*; Σ is a set of *letters*, an *alphabet*; $\hat{Q}$ is a non-empty set of *initial* states; $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*; and F is a non-empty set of *final* states.

A *word* over Σ is a (possibly empty) sequence of letters from Σ. For a word w, its *length* $|w|$ is the number of letters in w if w is finite and ω if w is infinite. We assume the standard definition of a *run* of the automaton on a word. If w is finite, a run on w is *accepting* if the last state of the run is in F; if w is infinite, a run is accepting by the Büchi condition if a state in F occurs on the run infinitely often. The *language* of an automaton is the set of words for which there exists an accepting run. One typically distinguishes between the finite-word language and the infinite-word language of an automaton. An automaton is *deterministic* if there is exactly one initial state and for every q and a, there is at most one q' such that (q, a, q') is in δ.

We use the standard abbreviations DFA, NFA and NBA for a deterministic automaton, a nondeterministic automaton over finite words, and a nondeterministic Büchi automaton over infinite words respectively.

LTL Linear Temporal Logic (LTL) is a logic defined over a set of *atomic propositions*, *AP*. The logic has the following minimal grammar, where $p \in AP$:

$$f := p \mid f_1 \wedge f_2 \mid \neg f_1 \mid \mathsf{X} f_1 \mid f_1 \,\mathsf{U}\, f_2$$

The satisfaction relation is defined over infinite words where each letter is a subset of AP. It has the form $w, i \models f$ for a word w and a natural number i, and is given by structural induction on formulas. We omit the standard definition. The language of a formula is the set of words that satisfy it. Standard constructions compile an LTL formula to an NBA that accepts the same language (cf. [18,39]), possibly incurring an exponential blowup.

Programs as Transition Systems A program is represented by its state transition system. This is a Moore machine, defined as a tuple $(S, \hat{S}, I, O, R, o)$ where S is a set of *states*, $\hat{S}$ is a non-empty set of *initial* states; I is a set of *input* values; O is a set of *output* values; $R \subseteq S \times I \times S$ is the *transition relation*, which must be total on I; and $o : S \to O$ is the *output mapping*. An *execution* of this system is an unbounded alternating sequence of states and inputs, and takes the form $s_0, a_0, s_1, a_1, \ldots$, such that for each i, the triple (s_i, a_i, s_{i+1}) is in the transition relation. A *computation* is an execution of this form where s_0 is an initial state.

Input-Output Words An input-output word (i/o word for short) is a pair of sequences (a, b) where a is a sequence of inputs, b is a sequence of outputs, and $|b| = 1 + |a|$. The input-output word induced by a program execution $s_0, a_0, s_1, a_1, \ldots$ is the pair (a, b) with $b = o(s_0), o(s_1), \ldots$. We sometimes write an i/o word in the linear format $b_0, a_0, b_1, a_1, \ldots$ for clarity. It is also common (cf. [32]) to view an infinite i/o word (a, b) in the "zipped" form $a \bowtie b = (a_0, b_0), (a_1, b_1), \ldots$. For a temporal property φ defined over input and output predicates and program M, the program M satisfies φ, written $M \models \varphi$, if the zipped input-output word of every computation of M satisfies φ. Each atomic proposition is a function in $I \times O \to \mathsf{Bool}$; an i/o pair (a, b) induces the set of propositions $\{p \mid p(a, b)\}$.

Games and Strategies A *strategy* is a function from finite sequences of inputs to outputs, represented as $\sigma : I^* \to O$. For an infinite input sequence $a = a_0, a_1, \ldots$, the strategy σ induces the infinite output sequence denoted $\sigma(a)$, given by $\sigma(\epsilon), \sigma(a_0), \sigma(a_0, a_1), \ldots$. A *play* for input a is the i/o word $(a, b = \sigma(a))$. We sometimes abuse this notation and use $\sigma(a)$ to refer to the play induced by a. A play is *winning* for a temporal property φ if it satisfies this property when viewed as a zipped i/o word. A strategy σ is *winning* for φ if for every input a, the play on a is winning for φ.

The *realizability* question is: given a property φ, determine whether there *exists* a program satisfying φ. The *synthesis* question is: given a property φ that is realizable, *construct* a program that satisfies φ. A strategy σ induces a *deterministic* program with an infinite state space, denoted $P(\sigma) = (S, \hat{S}, I, O, R, o)$. The state space S is the set of finite input sequences I^*, the initial state is ϵ, the output label for state x is $\sigma(x)$ and the transition relation R is given by

$\{(x, a, xa) \mid x \in I^*, a \in I\}$. This is in fact an infinite complete tree over I (sometimes called a "fulltree") where each node is labeled by an output value. A labeled fulltree, in turn, corresponds to a strategy and a deterministic program.

Synthesis Methods for Temporal Properties There is an extensive literature on methods to synthesize programs from LTL specifications (cf. [34,32,31,26] and tools that implement various algorithms (cf. [33,8,17,27,35,20]), all based on the conversion from LTL formulas to equivalent automata.

The classical approach to realizability of temporal properties (which we only sketch here, cf. [34,32]) is via the connection between programs, strategies, and labeled fulltrees. If a property φ is realizable, there is a deterministic program M satisfying φ. This program may also be seen as a strategy and a fulltree. From a deterministic word automaton with the same language as φ, one constructs a tree automaton that accepts precisely the fulltrees that satisfy φ. Now φ is realizable if and only if the language of this tree automaton is non-empty. For properties in LTL, this procedure can be carried out in 2EXPTIME in the length of the formula φ; the problem is 2EXPTIME-complete [32]. A winning strategy can be extracted as a finite state, deterministic reactive program from the tree automaton, thanks to the finite-model property of temporal logic. This approach is implemented in the tool Strix [30].

Two other approaches have been developed. One is to limit the logic: the GR(1) fragment expresses many useful properties, has a lower complexity (DEXPTIME), and can be implemented easily using symbolic (BDD-based) methods [31]. This is implemented in several tools [33,8,17,27]. The *bounded synthesis* method applies to full LTL and is iterative in nature. By placing bounds on the size of the intended program and the ranking argument for formula satisfaction, one obtains a simpler safety game, which can be solved using symbolic methods [26,35,20]. The approach is implemented in [11,19].

We use two of the approaches described above in this work. The classical approach is used to determine compact realizability of an arbitrary LTL formula, while GR(1) approach is used in the multi-robot setting.

3 Compactness

We formulate compactness for temporal specifications, investigate its properties, and show how to synthesize a compact strategy through a specification transformation. We consider specifications on infinite words for simplicity and to match the semantics of temporal logic.

A relation $\prec_O$ over the set of infinite output words is a *preference* relation if its transitive closure $\prec_O^+$ is irreflexive. We informally say that word b is better than b' if $b \prec_O^+ b'$ holds. As the transitive closure is irreflexive, it is not possible for a word to be better than itself, matching intuition. This relation is extended to input-output words as follows. An i/o word (a, b) is better than an i/o word (a', b') if (1) the input sequences a and a' are identical, and (2) $b \prec_O^+ b'$. The first condition ensures that comparable words have the same input sequence, which

is important as we are ultimately interested in the i/o words that are generated as plays of strategies.

Definition 1 (Compact Strategy). *A strategy σ is* compact *for an i/o language L if (1) σ is a winning strategy for L and (2) for every input sequence a, there is no i/o word (a, b') such that (a, b') satisfies L and (a, b') is better than the i/o word $(a, b = \sigma(a))$ that is produced as the play of σ on input a.*

The first condition ensures that σ is a valid strategy for L; the second that a compact strategy produces the "best possible" output for each input. We say that a language L is *compactly realizable* if it has a compact strategy.

Theorem 1. *A language L is realizable if it is compactly realizable. The converse does not hold.*

Proof. From right-to-left, consider a compact strategy σ for L. From the definition, σ is a winning strategy for L, hence L is realizable.

The converse does not hold. Let the input set $I = \{0, 1\}$ and the output set $O = \{c, d\}$ with the output preference ordering $c < d$ extended point-wise to output words. Let the specification L consist of sequences of the form $c(0c)^\omega$ and $d(\{0, 1\}d)^\omega$. This is realizable. No winning strategy can produce c on ϵ as there can be no win on input 1^ω. The single winning strategy produces d on every input sequence, including ϵ. But this strategy is not compact: for input 0^ω it generates $d(0d)^\omega$, but there is the better word $c(0c)^\omega$ in L. □

Standard realizability is monotone: if $L' \subseteq L$ and program M satisfies L', then M also satisfies L. However, compact realizability is neither monotone nor anti-monotone (proof in the full version). As is the case with deduction systems for commonsense reasoning (cf. [37]), non-monotonicity is a consequence of the formulation in terms of minimality.

The simple example from the Introduction is easily extended to a collection of N "if-condition-then-action" requirements. The IFTTT service (https://ifttt.com) or Apple Shortcuts implement these operationally, using an event-driven rule engine. However, from the viewpoint of temporal logic and synthesis, the results can be unexpected, as we have seen. The N requirements in LTL have the shape $(\bigwedge i : \mathsf{G}(a(i) \;\Rightarrow\; \mathsf{X}\, b(i)))$. The smallest model is one with a single state, issuing all the b actions unconditionally. This is clearly unintended. The intended model, which is compact, has 2^N states, one for each subset of the b actions. Thus, the gap between the smallest non-compact and compact models can be exponential in the length of the specification.

We now show the main theorem that links compact and standard realizability through a specification transformation.

Definition 2 (minimal language). *For a language L over alphabet Σ and a preference relation $\prec$ on Σ-words, the minimal elements of L form the language*

$$\min(L, \prec) = \{x \mid x \in L \wedge \neg(\exists y : y \in L \wedge y \prec^+ x)\}$$

I.e., a word x is in $\min(L, \prec)$ if it belongs to L and there is no word y in L that is transitively better than x.

Theorem 2. *Language L is compactly realizable if and only if* $\min(L, \prec)$ *is realizable.*

Proof. (Left-to-right) Let σ be a strategy that compactly realizes L. Consider any input sequence a. The output $b = \sigma(a)$ produced by the strategy is such that there is no word in L that is better than (a, b), by the definition of compactness. Hence, (a, b) is in $\min(L, \prec)$. As this holds for each input sequence, σ is a winning strategy for $\min(L, \prec)$.

(Right-to-left) Let σ be a winning strategy for $\min(L, \prec)$. For any input sequence a and its corresponding output $b = \sigma(a)$, the word $x = (a, b)$ must satisfy $\min(L, \prec)$. By the definition of min, we have that (1) x also satisfies L. Moreover, (2) there is no i/o word y that is better than x and also satisfies L. From (1) and (2), σ is a compact strategy for L. □

3.1 Effective Minimality Constructions for LTL

Theorem 2 implies that one can reduce compact realizability to standard realizability. Given a temporal specification φ, we transform its language $\mathcal{L}(\varphi)$ to the language $\mathcal{C}(\varphi) = \min(\mathcal{L}(\varphi), \prec)$. Starting from an LTL formula f, we give two constructions: one for the minimal language $\mathcal{C}(f)$, the other for its complement. The constructions assume that the relation $\prec^+$ can be expressed as an NBA, which is the case for the preference order defined in the Introduction.

The first construction directly follows Definition 2. The left-hand term ($x \in \mathcal{L}(f)$) is fulfilled by the standard conversion from LTL formula f to an NBA $\mathcal{A}_f$. For the right-hand term, we use the same NBA $\mathcal{A}_f$, now re-defined over y, for the $y \in \mathcal{L}(f)$ term; intersect this with the NBA for $\prec^+$; then project onto x and complement to obtain an NBA for the right-hand conjunct. The intersection of these two NBAs provides an NBA for $\mathcal{C}(f)$. These steps may result a worst-case double exponential blowup in the size of f: the first exponential is in the construction of $\mathcal{A}_f$; the second is in the complementation step. A similar construction applies if the specification is given directly as an NBA.

The second construction produces an NBA for the *complement* of the minimal language, with "only" a worst-case *single* exponential blowup. The complement of $\mathcal{C}(f)$ is (from the definition) $\{x \mid x \notin \mathcal{L}(f) \vee (\exists y : y \in \mathcal{L}(f) \wedge y \prec^+ x)\}$. For an LTL formula f, one constructs NBAs $\mathcal{A}_f$ and $\mathcal{A}_{\neg f}$ for the LTL formulas f and $\neg f$, respectively. An NBA for $(\exists y : y \in \mathcal{L}(f) \wedge y \prec^+ x)$ is obtained as in the first construction by omitting the final complementation step. The union of this NBA with the NBA for $\mathcal{A}_{\neg f}$ gives an NBA for the complement of $\mathcal{C}(f)$.

The NBA for the complement of $\mathcal{C}(f)$ can be used to model-check whether a given strategy is compact. It can also be used to synthesize machines using bounded synthesis, which requires an NBA for the complement of the specification property. The worst-case blowups are unavoidable: that follows from a lower-bound result by Birget [5] and a simpler but less general result of ours, discussed in the full version of this paper.

3.2 Relationship to Quantitative Synthesis

The formulation of compactness is in terms of a *qualitative* notion of minimality. A natural question is the relationship to methods for synthesis with *quantitative* objectives; in particular, methods for producing programs with optimal worst-case or average-case behavior [6,13]. Expanding on the argument in the Introduction, we establish that worst-case optimality cannot always distinguish between compact and non-compact solutions to a given specification.

In quantitative formulations, the synthesis game is formulated so that each transition has an associated reward. The reward of an infinite computation is defined using standard cumulative metrics such as mean-payoff (the limit of average rewards over successively longer prefixes) or discounted sum (the sum of rewards over the computation discounted geometrically, i.e., the k'th reward contributes a factor d^k, where $d \in (0,1)$ is the discount factor). The objective is to find a winning strategy with maximum worst-case reward, where the worst-case reward is the minimum reward over all inputs. In the stochastic form of the game, an additional probabilistic player "Nature" is introduced, and the objective is to find a winning strategy with the maximum average-case reward, where the average is the expectation taken over the induced probability space. Precise definitions of these concepts can be found in [6].

Worst-case optimality We return to the example discussed in the Introduction. There, we had assumed for simplicity that each action set is assigned a cost that is its cardinality. However, the reasoning carries over to any cost function that is monotonic with respect to set inclusion: i.e., if $A \subset B$ then $\mathsf{cost}(A) < \mathsf{cost}(B)$. Intuitively, monotonicity captures the preference for choosing a smaller set of output actions. Consider the mean-payoff cost of an infinite execution where the input a is always true. For the non-compact program in Figure 1(i), it is obvious that the limit of the average cost is $\mathsf{cost}(\{b\})$. That is also the case for the compact program in Figure 1(ii): the fact that the initial cost is $\mathsf{cost}(\emptyset)$ is swamped in the limit. This is the worst case input for both programs by the monotonicity of the cost function. The best case for the program on the right is when the input a is almost everywhere false. Thus, worst-case optimality cannot distinguish between the two programs for *any* monotonic cost function.

Average-case optimality We now show that average-case optimality also cannot always distinguish between compact and non-compact strategies. The general principle is that if a strategy is non-compact only for a finite prefix of a computation, its average-case cost in the limit will be the same as the cost of a strategy which performs in a compact manner throughout.

Consider the input set $I = \{0, 1\}$. Suppose that inputs are chosen uniformly at random. The output set O is the set of subsets of the action set $A = \{a, b\}$. Let the specification be the following: the initial choice of output set is either $\{a\}$ or A; all subsequent outputs must be A. There are only two winning strategies, which differ only in their choice of initial output (either $\{a\}$ or A); both produce output A subsequently regardless of the input. Assuming unit cost per output

action, the average cost of a run of length n is thus $(1+2(n-1))/n$ for the first strategy and 2 for the second. In the limit, both strategies have average cost 2, although the first is compact, while the second is not. This argument also applies for an arbitrary but monotone cost function.

In our view, quantitative measures are best suited to modeling the real cost of actions rather than to modeling a preference ordering. The two may, however, be combined to good effect. As compactness is ensured with a specification transformation, one can modularly apply quantitative synthesis to the transformed specification $\min(L, \prec)$ to obtain strategies that are compact and also optimal with respect to a cost metric.

3.3 Approximating Compactness

The worst-case exponential blowups can make it difficult to produce compact strategies. Moreover, Theorem 1 asserts that there are specifications that are realizable but have no compact strategies. For both reasons, we describe methods by which one can approximate the compactness criterion.

Approximately Minimal Languages The first method is to tighten the language L to L' that lies between L and $\min(L, \prec)$; we call L' *approximately minimal* for L. We synthesize a program satisfying L'. Given an NBA $\mathcal{A}$ for L over alphabet $I \times O$, we construct an NBA $\hat{\mathcal{A}}$ whose language is approximately minimal for L. This construction applies only to a class of preference relations that are induced pointwise by a partial order $\leq$ on individual letters of the output set O.

For infinite i/o words $w = (a, b)$ and $w' = (a', b')$, define $w \prec_p w'$ iff (1) for all i, $a_i = a'_i$ (inputs are identical) and $b_i \leq b'_i$, and (2) there is some i for which $b_i < b'_i$. We say that $w \preceq_p w'$ if $w \prec_p w'$ or $w = w'$. The ordering $\prec_p$ is transitive and regular. It is easy to construct an automaton accepting $\prec_p$, which checks condition (1) at each position of the zipped word $w \bowtie w'$, and accepts only if condition (2) holds at some position on the zipped word. The subset and cardinality preference relations introduced earlier are of this type.

Given an NBA $\mathcal{A}$ recognizing L, the NBA $\hat{\mathcal{A}}$ is constructed by excluding certain transitions of $\mathcal{A}$. Specifically, a transition $(q, (a_1, b_1), q')$ of $\mathcal{A}$ is omitted in $\hat{\mathcal{A}}$ if there is a "better" transition $(q, (a_2, b_2), q')$ in $\mathcal{A}$ with $a_1 = a_2$ and $b_2 < b_1$. Automaton $\hat{\mathcal{A}}$ can be efficiently constructed from $\mathcal{A}$ by performing a single pass over δ. The set of states, initial states and final states are identical in $\mathcal{A}$ and $\hat{\mathcal{A}}$.

Theorem 3. *For a pointwise preference order $\leq$ over O, $\mathcal{L}(\hat{\mathcal{A}})$ is an approximately minimal language for L.*

Proof. It is easy to see that $\mathcal{L}(\hat{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$, as an accepting run in $\hat{\mathcal{A}}$ is also an accepting run in $\mathcal{A}$.

For the other inclusion, let w be in $\min(L, \prec_p)$. Then, w is also in L. Thus, there is an accepting run ρ for w in $\mathcal{A}$. If all transitions in ρ are present in $\hat{\mathcal{A}}$, then w is also in $\mathcal{L}(\hat{\mathcal{A}})$. If not, there is a transition $(q, (a_1, b_1), q')$ at the k-th

step of ρ (for some k) that is not present in $\hat{\mathcal{A}}$. By construction, there must be a transition $(q, (a_1, b_2), q')$ in $\mathcal{A}$ such that $b_2 < b_1$. Now consider the run ρ' that is generated by swapping transition $(q, (a_1, b_1), q')$ with $(q, (a_1, b_2), q')$ at the k-th step. This is also an accepting run, on a word w' that is identical to w except that it has (a_1, b_2) rather than (a_1, b_1) as its k-th entry. As w' is in L and $w' \prec_p w$, it cannot be the case that w is in $\min(L, \prec_p)$, a contradiction. $\square$

Minimal Strategies for *L* The second method searches greedily for compact strategies in a game graph for L. For strategies σ and σ', say that $\sigma \sqsubseteq \sigma'$ (read as "σ is better than σ'") if for all input sequences a, $\sigma(a) = \sigma'(a)$ or $\sigma(a) \prec^+ \sigma'(a)$. I.e., the output on input a is using σ is at least as good as that using σ'. The minimal elements according to this ordering are called *minimal strategies* for L. It is easy to show that every compact strategy for L is a minimal strategy for L. The converse does not hold.

The greedy construction applies to a game graph for L where strategies are memoryless (e.g., if synthesis for L is a safety game or a parity game), and if the preference order is pointwise, as defined above. The core idea is simple: compute the set of winning positions; then nondeterministically and greedily extract a strategy by choosing only those transitions between successive winning positions that are output-minimal with respect to $<$. In the full version, it is shown that any strategy extracted in this manner is minimal for L.

4 Evaluation

4.1 Multi-Robot Coordination

Our original motivation to investigate compactness comes from an application to multi-robot orchestration. Due to space limitations, we describe this setting in brief. One has available multiple, heterogeneous robots, each capable of carrying out certain actions, some of which cannot be allowed to overlap. The goal is to perform specified tasks by (a) assigning robots to carry out actions and (b) sequence the actions appropriately. Tasks are described in a simple declarative language, called Resh [12], that has been implemented and used to control groups of mobile robots. A useful subset of Resh is given by the following grammar, where A is the set of action names and R is a set of robot names.

$$S := a \rightarrow R \mid S \Rightarrow S \mid S \,\&\, S \mid S | S \mid S + S$$

The interpretation of these operators is in terms of a *finite-word* input-output sequence. A term $a \rightarrow R$ is interpreted as "perform action a using one of the robots in R." For this, a control strategy chooses a robot r in R, and produces a "(begin a on r)" *output* event. Action duration is not fixed: E.g., the time taken to perform a "move to position p" action may vary as the robot maneuvers around humans. The completion signal is a "(end a on r)" event that is an *input*

to the control strategy. The other operators are interpreted as $\Rightarrow$ (sequencing), & (concurrent), | (choice), and + (concurrent with both tasks starting together). The interpretation of each operator produces a regular language.

The finite-word semantics is appropriate for robotics tasks that must be performed to completion. The same observation motivates the use of $\mathsf{LTL}f$ (a finite-word variant of LTL) in [38] to specify robotics tasks. A winning control strategy is one that satisfies the semantics of the operators.

As action completions are uncontrolled, even a simple specification such as $a \rightarrow R$ is unrealizable if the completion signal is never issued by an adversarial environment. It is thus necessary to restrict the environment so that every initiated action is eventually completed. This assumption *must* be interpreted over infinite words. It has the shape of a conjunction of LTL formulas $\mathsf{G}(\text{begin}(a, r) \Rightarrow \mathsf{X}\,\mathsf{F}\,\text{end}(a, r))$ over all actions a and robots r. This can be represented by a DBA which tracks the set of pending (i.e., begun but not ended) actions. This DBA is worst-case exponential in the number of action-robot pairs, but in practice is limited by the concurrency in the specification.

In order to match the infinite-word environment constraint, the Resh system specification must be extended to infinite words. This is done by saying that an infinite word w satisfies the specification if there is a prefix x of w such that x satisfies the specification. Being a regular language, a Resh specification is representable as a DFA; this is extended to infinite words as a DBA by replacing the outgoing transitions of each final state with a self-loop on all inputs.

We have arrived at the final form of the synthesis question, which has the shape $\mathcal{E} \Rightarrow \mathcal{S}$, where $\mathcal{E}$ (the environment assumptions) and $\mathcal{S}$ (the system specification) are both representable as deterministic Büchi automata. That is precisely the general form of a GR(1) specification [31]; therefore, algorithms for GR(1) synthesis can be applied to synthesize finite-state controllers.

Implementation and Experiments. Our initial experiments in synthesis with $(\mathcal{E} \Rightarrow \mathcal{S})$ occasionally produced non-compact strategies, which motivated this exploration of compactness. We now use the modified specification $\mathcal{E} \Rightarrow \hat{\mathcal{S}}$, where the system portion is made approximately compact through the construction in Section 3.3, which preserves the GR(1) format. This specification produces compact strategies for all cases we have examined.

Our implementation of GR(1) synthesis uses a SAT solver, similar to the method of [10]; we found this to be significantly faster than BDD-based methods. As there is not a well-defined set of benchmarks for robotics or Resh specifications, we generate 500 specifications at random, producing specifications with parse-tree depth 4, biased slightly to prefer the sequencing operation (i.e., $\Rightarrow$) over the others, as is likely to be the case in practice.

The system specification is set up to have two robots. Actions are allowed to overlap, which implies that all specifications are realizable. Of the 500 specifications, the GR(1) game graph was generated for 428 (85%) within a timeout limit of 5 minutes for each specification. (The Resh-to-automaton construction uses BDDs to symbolically represent output event sets, which sometimes blows up.) All 428 game graphs are solved by the SAT-based GR(1) procedure within

a timeout of 5 minutes per game. The median solution time is 3 seconds; 90% are solved within 30 seconds; and all are solved within 225 seconds. We also experimented with a small hand-designed group of specifications where certain action overlaps are forbidden, which are also resolved efficiently.

4.2 Compactness for LTL

We now describe an implementation of a compact synthesis pipeline for general LTL specifications. Our experiments use the benchmarks from the SYNTCOMP (2020) competition.[4] In these experiments, the preference order is fixed as the pointwise subset order. We were forced to make this arbitrary choice as there is limited information about the origin of the benchmark problems, so we could not tailor the ordering to the problem domain.

The goal is (1) to determine the difficulty of constructing a compact synthesis pipeline for LTL, and (2) to gauge the practical feasibility of the compact synthesis procedures. The experiments are designed to answer the following questions that arise from (2): **(Q1)** What is the overhead on generating compact strategies compared to standard synthesis? **(Q2)** Is the approximation procedure more efficient than exact compactness? and **(Q3)** How effective are the approximate constructions at producing compact strategies?

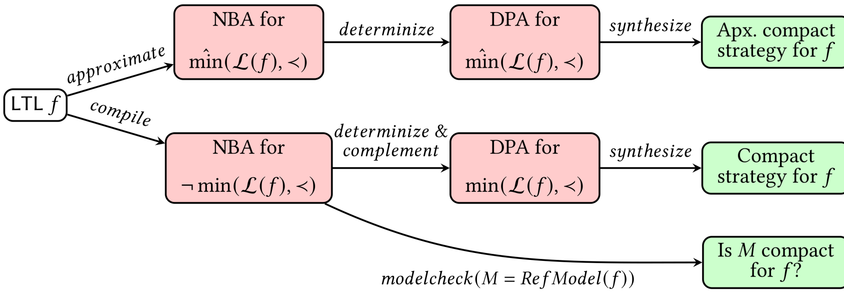

Fig. 2. An overview of the workflow for our experiments and tool. In the figure, $\hat{\min}$ refers to the approximate minimal language, while $RefModel(f)$ refers to the reference model for formula f.

A high-level overview of the internal structure of our tool is in Fig. 2. Our implementation chains together several known tools: the automaton libraries SPOT (v. 2.9.5) and Owl (v.20.06)[25], the synthesis tool Strix (v. 20.10)[30] and the model checker NuSMV (v. 2.6.0) [14]. We also use the AIGER toolkit [4] as well as the Syfco synthesis format converter[5]. We are grateful to the authors for making these tools freely available.

[4] At https://github.com/SYNTCOMP/benchmarks.
[5] At https://github.com/reactive-systems/syfco

Our tool offers three main features: (1) **Compact Realizability:** given an LTL formula f, determine if f is compactly realizable. This feature uses the compactness transformation from Section 3.1 to produce an automaton for the complement of the minimal language, which is then complemented, determinized and synthesized using Strix; (2) **Compactness Test:** given an LTL formula f and a candidate program P, determine if P is a compact program for f; and (3) **Approximate Compact Realizability:** given an LTL formula f, generate an approximately compact strategy. Here we implement the construction of the approximate minimal automaton from Section 3.3.

Our experiments were carried out on a Linux VM running Ubuntu 20.04 with 12 GB of memory. Naturally, we only consider synthesizing compact strategies for specifications that are realizable [6]. The results can be summarized as follows: **(A1)** We compare the efficiency of compact synthesis to the standard synthesis by evaluating the number of specifications that can be synthesized within a certain time limit. We fix this time limit to be 10 minutes, and use Strix for standard synthesis. (With this limit, the entire run over the benchmarks takes several hours.) Strix determines realizability for 396 specifications out of 421 ($\sim 94\%$), while our tool determines *compact* realizability for 213 ($\sim 50\%$). **(A2)** Within the same time limit, the approximation technique determines realizability for 398 specifications, significantly more than for exact compactness and about the same as for standard realizability. **(A3)** We model-check the strategies generated through approximate compact realizability. Model-checking for compactness requires the complement minimal automaton of a specification, so we set the time limit of 10 minutes per specification to generate this automaton. Within this limit, our tool manages to construct the required automaton for 246 specifications. Generating approximate compact strategies for these 246 specifications, and applying the Compactness Test on these strategies, we find that $\sim 42\%$ of the synthesized strategies are compact.

In addition, we tried our tool on the generalized version of the example specification from the introduction ($\bigwedge i : \mathsf{G}(a(i) \Rightarrow \mathsf{X}\, b(i))$). Our tool can synthesize a compact strategy till $N = 8$ fairly quickly, after which our setup struggles to compile the original LTL formula to an NBA. On the same set of specifications, the approximate techniques also produce a compact strategy.

The implementation process was fairly straightforward, a pleasant surprise given the number of tools and format conversions involved. We had to patch some tools to extend their capabilities (e.g., to allow automata as specifications) and to implement format conversions.

In summary, compact synthesis is feasible for a substantial number of specifications. Where it is not – due either to blowups in automaton construction or due to the gap between normal and compact realizability – one can use the approximation procedure defined in Section 3.3 to generate strategies that are minimal with respect to the strategy ordering.

[6] We refer to the helpful classification of these benchmarks into realizable and unrealizable ones from https://github.com/meyerphi/syntcomp-reference/.

5 Related Work

We discuss closely related work in synthesis and commonsense reasoning.

Qualitative Temporal Synthesis There is a considerable literature on the synthesis of open reactive programs from LTL specifications, starting with the seminal work by Pnueli and Rosner [32]. The beautiful theoretical results are made practical by the discovery of efficient algorithms for the GR(1) subclass [9,31], and procedures for bounded synthesis [21,36], based on so-called "Safraless" procedures [26]. These algorithms have been implemented in several tools, e.g., [11,15,16,19,23,33,27]. Our work builds on this basis by transforming the search for compact strategies to a standard synthesis question that can be handled by these tools.

In the robotics domain, prior work investigates synthesis for an interpretation of LTL over finite words called LTLf [22,38,40]. Although Resh is similarly restricted to finite-word properties, a central difference is that specifications in LTLf (like LTL) are defined over propositions on robot and world state, and not in terms of actions of an unknown duration.

There are many ways to choose between satisfying models: e.g., [7] designs synthesis procedures that produce minimally vacuous models. While the formulations differ, there is a common thread in the notion of minimality with respect to an ordering over models.

Quantitative Temporal Synthesis A substantial body of work in temporal synthesis is focused on *quantitative* objectives. These problems are represented by games where each action has an associated cost (or, dually, reward) and the objective is to find strategies that minimize cost (or, maximize reward) (cf. [6]). There are several ways to formulate appropriate cost/reward functions and correspondingly many ways to solve such games. One could attempt to model compactness by assigning costs to actions such that if word x is better than word y then x has the lower cost. We chose not to develop solutions along such quantitative lines for two main reasons: first, as the connection between cost and preference is indirect, setting up the right cost assignments to model a desired preference ordering is difficult; secondly, the theoretical complexity and practical difficulty of quantitative synthesis is high. Instead, we chose to tackle the question in a qualitative manner.

As shown in Section 3, quantitative measures cannot always differentiate between compact and non-compact solutions. Using the specification transformation developed here, the two methods can, however, be used in cooperation: one can model the real costs of actions in a manner that is orthogonal to the preference ordering and compute minimal-cost, compact strategies.

A recent work [1] focuses on the "quality" of satisfaction of an LTL formula (e.g., preferring to satisfy one part of a specification over another). Synthesis is through a reduction to a standard LTL specification; unfortunately this has a worst-case exponential blowup.

Non-Monotonic Reasoning. As mentioned briefly in the introduction, the compactness criterion is a form of commonsense reasoning: one does not expect synthesized solutions to include unnecessary actions. Commonsense reasoning is exemplified by the classical *frame problem*, introduced in [29], which shows that the freedom of interpretation given by logic must be restricted in order to achieve commonsense conclusions.

It was soon recognized that such restrictions imply a non-standard notion of deduction, which is not monotonic: adding new hypotheses can invalidate current conclusions [37]. In [28], McCarthy suggests a formulation in terms of a *circumscription* operation: each inference is guarded with a “not(abnormal)” predicate, and a successor state is one where the extent of this predicate is minimized—i.e., abnormal effects are maximally limited while avoiding inconsistencies. Logically, this is specified in second-order logic as $\varphi(A) \wedge \neg(\exists B : B \subset A \wedge \varphi(B))$, where φ is the specification and A is the abnormality predicate. Readers will immediately notice the similarity to the definition of $\min(L, \prec)$.

The importance of a general preference order in place of the fixed subset relation is laid out in [24]; the authors propose reasonable properties that any non-monotonic inference relation should meet, and show that a definition in terms of a preference ordering satisfies those properties. Our formulation of compactness is based on similar notions of minimality over a preference ordering on words. This is at the root of the non-monotonicity of compactness. These similarities hint at deeper connections between compactness and non-monotonic commonsense reasoning; we aim to investigate those in future work.

Acknowledgments. We would like to thank colleagues at Bell Labs and at NYU for interesting discussions and helpful comments regarding this work. This work was supported in part by the National Science Foundation grant CCF-1563393 and by DARPA under contract HR001120C0159. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense, the U.S. Government, or the National Science Foundation.

References

1. Almagor, S., Boker, U., Kupferman, O.: Formally reasoning about quality. J. ACM **63**(3), 24:1–24:56 (2016), https://doi.org/10.1145/2875421
2. Bansal, S., Chaudhuri, S., Vardi, M.Y.: Comparator automata in quantitative verification. In: FOSSACS. Lecture Notes in Computer Science, vol. 10803, pp. 420–437. Springer (2018)
3. Bansal, S., Namjoshi, K.S., Sa’ar, Y.: Synthesis of coordination programs from linear temporal specifications. Proc. ACM Program. Lang. **4**(POPL), 54:1–54:27 (2020)
4. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
5. Birget, J.: Partial orders on words, minimal elements of regular languages and state complexity. Theor. Comput. Sci. **119**(2), 267–291 (1993)

6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 140–156. Springer (2009)
7. Bloem, R., Chockler, H., Ebrahimi, M., Strichman, O.: Synthesizing non-vacuous systems. In: VMCAI. Lecture Notes in Computer Science, vol. 10145, pp. 55–72. Springer (2017)
8. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY - A new requirements analysis tool with synthesis. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 425–429. Springer (2010)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012)
10. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) VMCAI. LNCS, vol. 8318, pp. 1–20. Springer (2014)
11. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Proc. of CAV. pp. 652–657 (2012)
12. Carroll, M., Namjoshi, K.S., Segall, I.: The Resh programming language for multirobot orchestration. In: 2021 IEEE International Conference on Robotics and Automation, ICRA. IEEE (2021), at https://arxiv.org/abs/2103.13921
13. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: QUASY: quantitative synthesis tool. In: TACAS. LNCS, vol. 6605, pp. 267–271. Springer (2011)
14. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model verifier. In: CAV. LNCS, vol. 1633, pp. 495–499. Springer (1999), https://nusmv.fbk.eu/
15. Ehlers, R.: Symbolic bounded synthesis. In: Proc. of CAV. pp. 365–379 (2010)
16. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Proc. of TACAS. pp. 272–275 (2011)
17. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: CAV. Lecture Notes in Computer Science, vol. 9780, pp. 333–339. Springer (2016), https://github.com/VerifiableRobotics/slugs
18. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. Elsevier and MIT Press (1990). https://doi.org/10.1016/b978-0-444-88074-1.50021-4
19. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An experimentation framework for bounded synthesis. In: Proc. of CAV. pp. 325–332 (2017)
20. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: Proc. of CAV. pp. 263–277 (2009)
21. Filiot, E., Jin, N., Raskin, J.: Compositional algorithms for LTL synthesis. In: Proc. of ATVA. pp. 112–127 (2010)
22. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 854–860. IJCAI/AAAI (2013)
23. Jobstmann, B., Roderick: Optimizations for LTL synthesis. In: Proc. of FMCAD. pp. 117–124 (2006)
24. Kraus, S., Lehmann, D., Magidor, M.: Nonmonotonic reasoning, preferential models and cumulative logics. Artif. Intell. **44**(1-2), 167–207 (1990)
25. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: ATVA. LNCS, vol. 11138, pp. 543–550. Springer (2018), https://owl.model.in.tum.de/

26. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. of FOCS. pp. 531–540. IEEE, IEEE (2005)
27. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Softw. Syst. Model. **20**(5), 1553–1586 (2021). https://doi.org/10.1007/s10270-021-00868-z
28. McCarthy, J.: Circumscription - A form of non-monotonic reasoning. Artif. Intell. **13**(1-2), 27–39 (1980)
29. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969)
30. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018), https://strix.model.in.tum.de
31. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. In: International Conference on VMCAI. vol. 3855, pp. 364–380. Springer, Springer (2006)
32. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Prof. of POPL. pp. 179–190 (1989)
33. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A framework for developing verification algorithms. In: Proc. of CAV. pp. 171–174 (2010)
34. Rabin, M.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. (141), 1–35 (1969)
35. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: ATVA. Lecture Notes in Computer Science, vol. 4762, pp. 474–488. Springer (2007)
36. Schewe, S., Finkbeiner, B.: Bounded synthesis. Proc. of ATVA pp. 474–488 (2007)
37. Strasser, C., Antonelli, G.A.: Non-monotonic Logic. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, summer 2019 edn. (2019)
38. Tabajara, L.M., Vardi, M.Y.: Partitioning techniques in LTLf synthesis. In: IJCAI. pp. 5599–5606. ijcai.org (2019)
39. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 133–191. Elsevier and MIT Press (1990), https://doi.org/10.1016/b978-0-444-88074-1.50009-3
40. Zhu, S., Giacomo, G.D., Pu, G., Vardi, M.Y.: LTLf synthesis with fairness and stability assumptions. In: AAAI. pp. 3088–3095. AAAI Press (2020)

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.