Antichains and Compositional Algorithms for LTL Synthesis*

Emmanuel Filiot · Naiyong Jin · Jean-François Raskin

Abstract In this paper, we present new monolithic and compositional algorithms to solve the LTL realizability problem. Those new algorithms are based on a reduction of the LTL realizability problem to a game whose winning condition is defined by a universal automaton on infinite words with a k-co-Büchi acceptance condition. This acceptance condition asks that runs visit at most k accepting states, so it implicitly defines a *safety game*. To obtain efficient algorithms from this construction, we need several additional ingredients. First, we study the structure of the underlying automata constructions, and we show that there exists a partial order that structures the state space of the underlying safety game. This partial order can be used to define an efficient *antichain algorithm*. Second, we show that the algorithm can be implemented in an *incremental way* by considering increasing values of k in the acceptance condition. Finally, we show that for large LTL formulas that are written as conjunctions of smaller formulas, we can solve the problem *compositionally* by first computing winning strategies for each conjunct that appears in the large formula. We report on

* This paper extends the results of the two following previous papers [10, 11] by the authors.

Work supported by the projects: (i) QUASIMODO (FP7- ICT-STREP-214755), Quasimodo: "Quantitative System Properties in Model-Driven-Design of Embedded", `http://www.quasimodo.aau.dk/`, (ii) GASICS (ESF-EUROCORES LogiCCC), Gasics: "Games for Analysis and Synthesis of Interactive Computational Systems", `http://www.ulb.ac.be/di/gasics/`, (iii) Moves: "Fundamental Issues in Modelling, Verification and Evolution of Software", `http://moves.ulb.ac.be`, a PAI program funded by the Federal Belgian Gouvernment, and (iv) ECSPER (ANR-JC09-472677) and SFINCS (ANR-07-SESU-012), two projects supported by the French National Research Agency.

E. Filiot
Département d'Informatique Université Libre de Bruxelles
Bld du Triomphe CP 212
1050 Brussels
Belgium
Tel.: +32-2-6506464
Fax: +32-2-650-56-09
E-mail: efiliot@ulb.ac.be

N. Jin
E-mail: naiyjin@ulb.ac.be

J.-F. Raskin
E-mail: jraskin@ulb.ac.be

the behavior of those algorithms on several benchmarks. We show that the compositional algorithms are able to handle LTL formulas that are several pages long.

Keywords LTL realizability and synthesis · Automata on infinite words · Compositional algorithms · Antichain algorithms.

1 Introduction

Context and motivations The *realizability problem* is best seen as a game between two players [24]. Given an LTL formula ϕ and a partition of its atomic propositions P into I and O, Player 1 starts by giving a subset $o_0 \subseteq O$ of propositions[1], Player 2 responds by giving a subset of propositions $i_0 \subseteq I$, then Player 1 gives o_1 and Player 2 responds by i_1, and so on. This game lasts forever and the outcome of the game is the infinite word $w = (i_0 \cup o_0)(i_1 \cup o_1)(i_2 \cup o_2) \cdots \in (2^P)^\omega$. Player 1 wins if the resulting infinite word w is a model of ϕ. The *synthesis problem* asks to produce a winning strategy for Player 1 when the LTL formula is realizable.

The LTL realizability problem is central when reasoning about specifications for reactive systems and has been studied starting from the end of the eighties with the seminal works by Pnueli and Rosner [24], and Abadi, Lamport and Wolper [1]. It has been shown 2EXPTIME-C in [26][2]. Despite their high worst-case computation complexity, we believe that it is possible to solve LTL realizability and synthesis problems in practice. We proceed here along recent research efforts that have brought new algorithmic ideas to attack this important problem.

The classical automata-based solution to LTL synthesis can be summarized as follows. Given an LTL formula ϕ, construct a nondeterministic Büchi automaton A_ϕ that accepts all models of ϕ, transform A_ϕ into a deterministic Rabin automaton B using Safra's determinization procedure [27], and use B as an observer in a turn-based two-player game. Unfortunately, this theoretically elegant procedure has turn out to be very difficult to implement. Indeed, Safra's determinization procedure generates very complex state spaces: states are colored trees of subsets of states of the original automaton. No nice symbolic data-structure is known to handle such state spaces. Moreover, the game to solve as the last step (on a potentially doubly-exponential state space) is a Rabin game, and this problem is known to be NP complete[3].

This situation has triggered further research for alternative procedures. Most notably, Kupferman and Vardi in [19] have recently proposed procedures that avoid the determinization step and so Safra's construction[4]. In particular, they reduce the LTL realizability problem to the emptiness of a Universal Co-Büchi Tree automaton (UCT). They show how to test emptiness of a UCT by translation to an alternating weak Büchi tree automaton, again translated into a non-deterministic Büchi tree automaton for which testing emptiness is easy. All these steps have been implemented and optimized in several ways by Jobstmann and Bloem

[1] Technically, we could have started with Player 2, for modeling reason it is conservative to start with Player 1. All the techniques developed in this paper can be trivially adapted to the other setting.

[2] Older pioneering works consider the realizability problem but for more expressive and computationally intractable formalisms like MSO, see [31] for pointers.

[3] Instead of Rabin automata, Parity automata can also be used [22]. Nevertheless, there are no known polynomial time algorithm to solve parity games.

[4] As a consequence, they call their new procedures *Safraless* procedures. Nevertheless they use the result by Safra in their proof of correctness.

in a tool called Lily [15]. In 2007, Schewe and Finkbeiner has shown how to reduce the emptiness problem of UCT into the emptiness of safety tree automata.

Contributions In this paper, our contributions are threefold. First, we phrase a Safraless decision procedure for the LTL realizability and synthesis problem directly in the formalism of infinite word automata. Second, we identify structural properties in the underlying automata constructions that allow us to define an antichain algorithm for solving the LTL realizability problem. This is in line with our previous works in [6,7,25,8] that use subsumption to obtain efficient implementations of several variants of subset constructions. Third, we study compositional algorithms to solve safety games, and we show how they can be used to develop compositional algorithms for solving the realizability and synthesis problems of large and structured LTL specifications.

Safraless procedure Our Safraless procedure uses Universal Co-Büchi Word automata, UCW. While solving the emptiness problem is easy for nondeterministic automata, solving the universality problem is easy for universal automata: a UCW A accepts all the words in Σ^ω if all cycles in A reachable from an initial state only contain non final states[5]. As a direct consequence, A is universal if and only if all paths starting from initial states in A can visit at most n times a final state, where n is the number of states in A. So universality easily reduces to safety for UCW. This simple property can be exploited in synthesis as follows: if a Moore machine M (representing a strategy) with m states defines a language (the outcome of the strategy) included in the language of a UCW A with n states, denoted by $L_{uc}(A)$, i.e., $L(M) \subseteq L_{uc}(A)$, then every run on the words generated by M contains at most $2mn$ final states of A. As a consequence, a strategy represented by a Moore machine that enforces a language defined by a UCW also enforces a *stronger specification* defined by the same automaton where the acceptance condition is strengthened to a so called "$2mn$-co-Büchi": a run is accepting if it passes at most $2mn$ times by a final state. Those automata are called *Universal k-co-Büchi automata*, denoted by UKCW. The language of A with this acceptance condition is denoted by $L_{uc,k}(A)$.

Using the result by Safra [27], we know that the size of a Moore machine that realizes a language defined by a UCW A can be bounded by some value $K \in \mathbb{N}$, which is at most exponential in the size of A. This gives a reduction from the general problem to the problem of the realizability of a UKCW specification. Contrarily to general UCW specifications, universal K-co-Büchi specifications are *safety conditions*, and they can easily be made deterministic. The ideas underlying our construction are similar to the ones used in the reduction from UCT to safety tree automata proposed in [28].

Antichain and incremental algorithm The realizability and synthesis problems of an LTL formula ϕ can thus be reduced to a game whose winning objective is expressed by a UCW A_ϕ, where A_ϕ is the UCW that accepts all the models of formula ϕ. The acceptance condition of this automata can be strengthened to a K-co-Büchi condition and made deterministic using an extension of the classical subset construction. When applied to a universal automaton A with set of states Q, the classical subset construction consists in building a new automaton A' whose states are subsets of Q. Thus, each state of A' encodes the set of states of A that are active at each level of the run tree. In the case of K-co-Büchi automata, one needs additionally to remember how many times final states have been visited on the

[5] Analysis of cycles in automata over infinite words has previously been exploited in bounded model-checking, see [18] for a formal treatment.

branches that lead to each active state. Clearly only the maximal number (up to $K+1$) of visits to final states among all the branches that reach q has to be remembered. So, we need one counter, that counts up to $K+1$, for each state of the automaton A_ϕ. To implement this approach in practice, we face two difficulties. First, the automaton A_ϕ can be exponentially larger than ϕ, and so its determinization can be doubly-exponentially larger than ϕ. Second, the maximal value $K \in \mathbb{N}$ that we need to consider in theory is also doubly exponential in the size of the formula ϕ. To overcome those two difficulties, we study the structure of the underlying automata constructions, and we develop the following two heuristics.

First, we show that the set of states of the deterministic automaton is partially ordered. The underlying partial order can be used to define an efficient data-structure to compactly represent and efficiently manipulate the game positions of the associated safety game. This allows us to develop an antichain algorithm, in the spirit of [8], to efficiently compute the winning positions in the safety game.

Second, for all UCW A, and for all $k_1, k_2 \in \mathbb{N}$, if $k_1 \leq k_2$ then $L_{uc,k_1}(A) \subseteq L_{uc,k_2}(A)$. So, instead of solving the safety game associated with the specification $L_{uc,K}(A_\phi)$ (for the theoretical bound K given by the Safra's construction), we adopt an incremental approach, and we solve the games underlying $L_{uc,i}(A_\phi)$ for increasing values of i, $i = 0, 1, 2, ..., K$. As soon as one of this game can be won by Player 1, we know that the formula is realizable because $L_{uc,i}(A_\phi) \subseteq L_{uc,K}(A_\phi) \subseteq L_{uc}(A_\phi)$. For unrealizable specification, this approach is not reasonable. This is why we consider in parallel the games associated with the specifications $L_{uc,i}(A_{\neg\phi})$ for increasing values of i, $i = 0, 1, 2, ..., K'$, and decide if Player 2 has a winning strategy in those games[6]. As LTL games are determined [21], we know that if Player 1 cannot realize ϕ, then Player 2 can realize $\neg\phi$. In practice, we will see that for all the LTL formulas that we consider in our benchmarks, one of the the specifications $L_{uc,i}(A_\phi)$ or $L_{uc,i}(A_{\neg\phi})$ is realizable for a small value of i (less than 3 in all our experiments). This incremental algorithm has been implemented in a prototype of tool called Acacia. We have applied it to a set of benchmarks provided with the tool Lily and compared the performances of the two approaches on those benchmarks.

Compositional algorithm Large LTL formulas are often written as conjunctions of smaller formulas. We show that if the LTL formula has the form $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, i.e., a conjunction of LTL sub-specifications, then $G(\Phi)$, the safety game underlying the formula Φ (as sketched above), can be constructed and solved compositionally. The compositional algorithms are able to handle formulas that are several pages long while non-compositional algorithms are limited to much smaller formulas.

The compositional algorithms rely on the following nice property of safety games: for any safety game G, there exists a function that maps each position s of Player 1 to the set of all actions that are safe to play in s. We call this function the *master plan* of Player 1 in G. It encompasses all the winning strategies of Player 1. If Λ is the master plan of G then we denote by $G[\Lambda]$ the game G where the behavior of Player 1 is restricted by Λ.

To compute the winning positions of a safety game $G^{12} = G^1 \otimes G^2$ defined as the composition of two sub-games, we compute the master plans for the local games G^1 and G^2 before composition. Let Λ_1 (resp. Λ_2) be the master plan for G^1 (resp. G^2), then the winning positions in G^{12} are the same as the winning positions in $G^1[\Lambda_1] \otimes G^2[\Lambda_2]$. We develop backward and forward algorithms that exploit this property.

Sometimes, the LTL formula is given in the following form: $\bigwedge_{i=1}^{m} \psi_i \rightarrow \bigwedge_{j=1}^{n} \phi_j$ where ψ_i are hypothesis that are made on the environment of the system to control, and ϕ_j are

[6] Note that in this game, Player 1 is first to play as in the original game.

guarantees that the controller has to ensure. For those formulas, we show how to rewrite them in order to apply the compositional algorithms and how to simplify the formula that we obtain after rewriting.

We have implemented the two compositional algorithms in our prototype Acacia and we provide an empirical evaluation of their performances on the set of benchmarks on which we have evaluated the monolithic incremental approach sketched before, and on a realistic case study taken from the IBM RuleBase tutorial [14].

Related works The first solution [24] to the LTL realizability and synthesis problem was based on Safra's procedure for the determinization of Büchi automata [27].

Following [19], the method proposed in our paper can be coined "Safraless" approach to the realizability and synthesis of LTL as it avoids the determinization (based on Safra's procedure) of the automaton obtained from the LTL formula. Our approach relies on a reduction to safety games, as in [28]. There, the construction is used to justify a reduction of the emptiness of universal co-Büchi tree automata to the SAT problem. In turn, this reduction is used to obtain a semi-algorithm for the distributed synthesis problem which is undecidable. Our algorithms are not reductions to SAT but fixed-point algorithms that can be implemented compositionally and symbolically using antichains. Recently, Ehlers [9] has also implemented this reduction with a fixed point algorithm using BDDs and not antichains.

In [19], Kupferman and Vardi proposed the first Safraless approach that reduces the LTL realizability problem to Büchi games, which has been implemented in the tool Lily [15], their algorithm is incremental as the algorithm proposed in that paper. In [17], a compositional approach to LTL realizability and synthesis is proposed. Their algorithm is based on a Safraless approach that transforms the synthesis problem into a Büchi and not a safety game as in our case. There is no notion like the master plan for Büchi games. To the best of our knowledge, their algorithm has not been implemented.

In [4], the idea of checking the realizability of ψ by Player 1 in parallel with the realizability of $\neg\psi$ by Player 2 is also proposed. Nevertheless, the procedure there is only complete for ω-regular specifications that are definable by deterministic Büchi automata.

In [23, 3], an algorithm for the realizability problem for a fragment of LTL, known as GR(1), is presented and evaluated on the case study of [14]. The specification into the GR(1) fragment for this case study is not trivial to obtain and so the gain in term of complexity[7] comes with a cost in term of expressing the problem in the fragment. Our approach is different as we want to consider the full LTL logic. In our opinion, it is important to target full LTL as it often allows for writing more declarative and more natural specifications.

In [29], the authors also consider LTL formulas of the form $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$. They propose an algorithm to construct compositionally a parity game from such LTL specifications. Their algorithm uses a variant of Safra's determinization procedure and additionally tries to detect local parity games that are equivalent to safety games (because the associated LTL subformula is a safety property). For efficiently solving the entire game, they use BDDs.

In [16], a compositional algorithm is proposed for reasoning about network of components to control under partial observability. The class of properties that they consider is safety properties and not LTL properties. They propose a backward algorithm and no forward algorithm.

The implementation supporting the approaches described in [29] and [3] uses BDDs while our tool Acacia does not. While our algorithms could have been implemented with BDDs (see [9] for such an implementation), we deliberately decided not to use them for two

[7] GR(1) has a better worst-case complexity than full LTL.

reasons. First, to fairly compare our Safraless approach with the one proposed in [19] and implemented in Lily, we needed to exclude BDDs as Lily does not use them. Second, several recent works on the efficient implementation of decision procedures based on variants of the subset construction show that antichain based algorithms may outperform BDD-based implementations by several orders of magnitude, see [6,7] for more details.

Outline The rest of the paper is structured as follows. Sect. 2 recalls the formal definitions of LTL, the realizability problem, universal automata operating on infinite words, and Moore machines to represent finite memory strategies. Sect. 3 recalls formal definitions for safety games, and two algorithms to solve them, one that operates backward and one that operates forward. Sect. 4 shows how the realizability problem of a ω-regular language defined by a universal automaton with co-Büchi acceptance condition can be reduced to the realizability problem of a language defined by the same automaton but with a K-co-Büchi acceptance condition. Sect. 5 uses this reduction to associate with each LTL formula a safety game. Sect. 6 studies the structure underlying this safety game to define an incremental antichain algorithm to solve the realizability problem. Sect. 7 evaluates the algorithms proposed in Sect. 6 on a large number of benchmarks. Sect. 8 introduces the compositional algorithms for solving the realizability problem of large LTL formulas given as conjunctions of smaller formulas. Sect 9 evaluates the compositional algorithms on the benchmarks of Sect. 7, and on a larger, scalable, and more realistic example.

2 LTL and Realizability Problem

*Linear Temporal Logic (*LTL*)* The formulas of LTL are defined over a set of atomic propositions P. The syntax is given by the grammar:

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid \mathcal{X}\phi \mid \phi \mathcal{U} \phi \qquad\qquad p \in P$$

The notations true, false, $\phi_1 \wedge \phi_2$, $\Diamond\phi$ and $\Box\phi$ are defined as usual. In particular, $\Diamond\phi = \mathsf{true}\,\mathcal{U}\phi$ and $\Box\phi = \neg\Diamond\neg\phi$. LTL formulas ϕ are interpreted on infinite words $w = \sigma_0\sigma_1\sigma_2\cdots \in (2^P)^\omega$ via a satisfaction relation $w \models \phi$ inductively defined as follows:

(i) $w \models p$ if $p \in \sigma_0$;
(ii) $w \models \phi_1 \vee \phi_2$ if $w \models \phi_1$ or $w \models \phi_2$;
(iii) $w \models \neg\phi$ if $w \not\models \phi$;
(iv) $w \models \mathcal{X}\phi$ if $\sigma_1\sigma_2\ldots \models \phi$, and (v) $w \models \phi_1\,\mathcal{U}\phi_2$ if there is $n \geq 0$ such that $\sigma_n\sigma_{n+1}\ldots \models \phi_2$ and for all $0 \leq i < n$, $\sigma_i\sigma_{i+1}\ldots \models \phi_1$.

Given a LTL formula ϕ, we note $[\![\phi]\!]$ the set of infinite words w s.t. $w \models \phi$.

LTL *Realizability and Synthesis* As mentioned in the introduction, the realizability problem for LTL is best seen as a game between two players. Each of the players is controlling a subset of the set P of propositions on which the LTL formula is constructed. Accordingly, unless otherwise stated, we partition the set of propositions P into I the set of *input signals* that are controlled by Player 2 (the environment), and O the set of *output signals* that are controlled by Player 1 (the controller). It is also useful to associate this partition of P with the three following alphabets: $\Sigma = 2^P$, $\Sigma_1 = 2^O$, and $\Sigma_2 = 2^I$. We denote by $\varnothing$ the empty set.

The realizability game is played in turns. Player 1 starts by giving a subset o_0 of propositions, Player 2 responds by giving a subset of propositions i_0, then Player 1 gives o_1 and Player 2 responds by i_1, and so on. This game lasts forever and the output of the game is the infinite word $(i_0 \cup o_0)(i_1 \cup o_1)(i_2 \cup o_2)\cdots \in \Sigma^\omega$.

The players play according to strategies. A strategy for Player 1 is a (total) mapping $\lambda_1 : (\Sigma_1\Sigma_2)^* \to \Sigma_1$ while a strategy for Player 2 is a (total) mapping $\lambda_2 : \Sigma_1(\Sigma_2\Sigma_1)^* \to \Sigma_2$. The outcome of the strategies λ_1 and λ_2 is the word $\mathsf{outcome}(\lambda_1, \lambda_2) = (o_0 \cup i_0)(o_1 \cup i_1)\ldots$ such that $o_0 = \lambda_1(\epsilon)$, $i_0 = \lambda_2(o_0)$ and for all $j \geq 1$, $o_j = \lambda_1(o_0 i_0 \ldots o_{j-1} i_{j-1})$ and $i_j = \lambda_2(o_0 i_0 \ldots o_{j-1} i_{j-1} o_j)$.

Given an LTL formula ϕ (the specification), the *realizability problem* is to decide whether there exists a strategy λ_1 of Player 1 such that for all strategies λ_2 of Player 2, $\mathsf{outcome}(\lambda_1, \lambda_2) \models \phi$. If such a strategy exists, we say that the specification ϕ is *realizable*. If an LTL specification is realizable, there exists a finite-state strategy that realizes it [24]. The *synthesis problem* is to compute a finite-state strategy that realizes the LTL specification.

Example 1 Let $I = \{q\}$, $O = \{p\}$ and $\psi = p\mathcal{U}q$. The formula ψ is not realizable. As q is controlled by the environment, he can decide to leave it always false and the outcome does not satisfy ϕ. However $\Diamond q \to (p\mathcal{U}q)$ is realizable. The assumption $\Diamond q$ states that q will hold at some point, and so, one of the possible winning strategies for Player 1 is to always assert p.

Infinite Word Automata An *infinite word automaton* over the finite alphabet Σ is a tuple $A = (\Sigma, Q, q_0, \alpha, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\alpha \subseteq Q$ is a set of final states and $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation. For all $q \in Q$ and all $\sigma \in \Sigma$, we let $\delta(q, \sigma) = \{q' \mid (q, \sigma, q') \in \delta\}$. We let $|A| = |Q| + |\delta|$ be the size of A. We say that A is *deterministic* if $\forall q \in Q \cdot \forall \sigma \in \Sigma \cdot |\delta(q, \sigma)| \leq 1$. It is *complete* if $\forall q \in Q \cdot \forall \sigma \in \Sigma \cdot \delta(q, \sigma) \neq \varnothing$. In this paper, unless otherwise stated and w.l.o.g., the automata are complete. A *run* of A on a word $w = \sigma_0\sigma_1 \cdots \in \Sigma^\omega$ is an infinite sequence of states $\rho = \rho_0\rho_1 \cdots \in Q^\omega$ such that $\rho_0 = q_0$ and $\forall i \geq 0 \cdot \rho_{i+1} \in \delta(\rho_i, \sigma_i)$. We denote by $\mathsf{Runs}_A(w)$ the set of runs of A on w. We denote by $\mathsf{Visit}(\rho, q)$ the number of times the state q occurs along the run ρ. We consider three acceptance conditions (a.c.) for infinite word automata. A word w is accepted by A if (depending on the a.c.):

Non-deterministic Büchi	:	$\exists \rho \in \mathsf{Runs}_A(w) \cdot \exists q \in \alpha \cdot \mathsf{Visit}(\rho, q) = \infty$
Universal Co-Büchi	:	$\forall \rho \in \mathsf{Runs}_A(w) \cdot \forall q \in \alpha \cdot \mathsf{Visit}(\rho, q) < \infty$
Universal K-Co-Büchi	:	$\forall \rho \in \mathsf{Runs}_A(w) \cdot \sum_{q \in \alpha} \mathsf{Visit}(\rho, q) \leq K$

The set of words accepted by A with the non-deterministic Büchi a.c. is denoted by $L_{\mathrm{b}}(A)$, and with this a.c. in mind, we say that A is a non-deterministic Büchi word automaton, NBW for short. Similarly, we denote respectively by $L_{\mathrm{uc}}(A)$ and $L_{\mathrm{uc},K}(A)$ the set of words accepted by A with the universal co-Büchi and universal K-co-Büchi a.c. respectively. With those interpretations, we say that A is a universal co-Büchi automaton (UCW) and that the pair (A, K) is a universal K-co-Büchi automaton (UKCW) respectively. By duality, we have clearly $L_{\mathrm{b}}(A) = \overline{L_{\mathrm{uc}}(A)}$, for any infinite word automaton A. Finally, note that for any $0 \leq K_1 \leq K_2$, we have that $L_{\mathrm{uc},K_1}(A) \subseteq L_{\mathrm{uc},K_2}(A) \subseteq L_{\mathrm{uc}}(A)$.

Infinite automata and LTL It is well-known (see for instance [30]) that NBWs subsume LTL in the sense that for all LTL formula ϕ, there is an NBW A_ϕ (possibly exponentially larger) such that $L_{\mathrm{b}}(A_\phi) = \{w \mid w \models \phi\}$. Similarly, by duality it is straightforward to associate

an equivalent UCW with any LTL formula ϕ: take $A_{\neg\phi}$ with the universal co-Büchi a.c., so $L_{\text{uc}}(A_{\neg\phi}) = \overline{L_{\text{b}}(A_{\neg\phi})} = \{w \mid w \not\models \neg\phi\} = \{w \mid w \models \phi\}$.

To reflect the game point of view of the realizability problem, we introduce the notion of turn-based automata to define the specification. A *turn-based automaton* A over the input alphabet Σ_2 and the output alphabet Σ_1 is a tuple $A = (\Sigma_2, \Sigma_1, Q_2, Q_1, q_0, \alpha, \delta_2, \delta_1)$ where Q_2, Q_1 are finite sets of input and output states respectively, $q_0 \in Q_1$ is the initial state, $\alpha \subseteq Q_2 \cup Q_1$ is the set of final states, and $\delta_2 \subseteq Q_2 \times \Sigma_2 \times Q_1$, $\delta_1 \subseteq Q_1 \times \Sigma_1 \times Q_2$ are the input and output transition relations respectively. It is *complete* if for all $q_2 \in Q_2$, and all $\sigma_2 \in \Sigma_2$, $\delta_2(q_2, \sigma_2) \neq \varnothing$, and for all $q_1 \in \Sigma_1$ and all $\sigma_1 \in \Sigma_1$, $\delta_1(q_1, \sigma_1) \neq \varnothing$. As for usual automata, in this paper we assume that turn-based automata are always complete. Turn-based automata still run on words from Σ^ω as follows: a run on a word $w = (o_0 \cup i_0)(o_1 \cup i_1)\cdots \in \Sigma^\omega$ is a word $\rho = \rho_0\rho_1\cdots \in (Q_1Q_2)^\omega$ such that $\rho_0 = q_0$ and for all $j \geq 0$, $(\rho_{2j}, o_j, \rho_{2j+1}) \in \delta_1$ and $(\rho_{2j+1}, i_j, \rho_{2j+2}) \in \delta_2$. All the acceptance conditions considered in this paper carry over to turn-based automata. Turn-based automata with acceptance conditions C are denoted by tbC, e.g. tbNBW. Every UCW (resp. NBW) with state set Q and transition set Δ is equivalent to a tbUCW (resp. tbNBW) with $|Q| + |\Delta|$ states: the new set of states is $Q \cup \Delta$, final states remain the same, and each transition $r = q \xrightarrow{\sigma_o \cup \sigma_i} q' \in \Delta$ where $\sigma_o \in \Sigma_1$ and $\sigma_i \in \Sigma_2$ is split into a transition $q \xrightarrow{\sigma_o} r$ and a transition $r \xrightarrow{\sigma_i} q'$.

Moore Machines LTL realizability is equivalent to LTL realizability by a finite-state strategy [24]. We use Moore machines to represent finite-state strategies. A *Moore machine* M with input alphabet Σ_2 and output alphabet Σ_1 is a tuple $(\Sigma_2, \Sigma_1, Q_M, q_0, \delta_M, g_M)$ where Q_M is a finite set of states with initial state q_0, $\delta_M : Q_M \times \Sigma_2 \to Q_M$ is a (total) transition function, and $g_M : Q_M \to \Sigma_1$ is a (total) output function. We extend δ_M to $\delta_M^* : \Sigma_2^* \to Q_M$ inductively as follows: $\delta_M^*(\epsilon) = q_0$ and $\delta_M^*(u\sigma) = \delta_M(\delta_M^*(u), \sigma)$. The language of M, denoted by $L(M)$, is the set of words $w = (o_0 \cup i_0)(o_1 \cup i_1)\cdots \in \Sigma_P^\omega$ such that for all $j \geq 0$, $\delta_M^*(i_0 \ldots i_{j-1})$ is defined and $o_j = g_M(\delta_M^*(i_0 \ldots i_{j-1}))$. In particular, $o_0 = g_M(\delta_M^*(\epsilon)) = g_M(q_0)$. The size of a Moore machine is defined similarly as the size of an automaton.

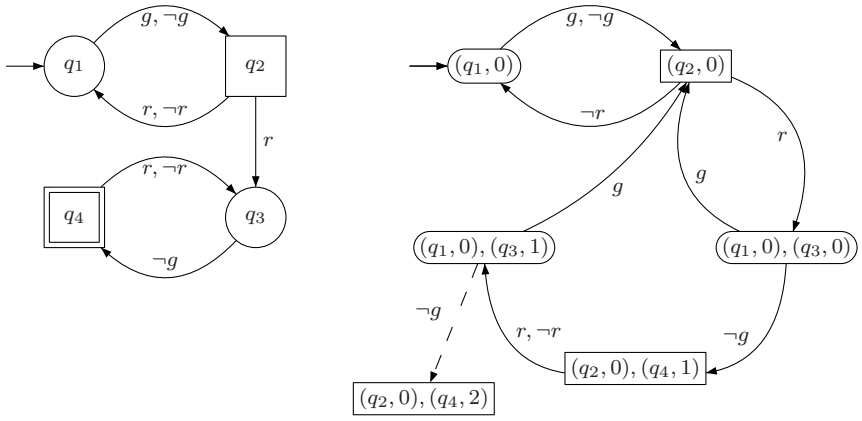
As for every LTL formula ϕ we can construct a tbUCW A_ϕ such that $L_{uc}(A_\phi) = [\![\phi]\!]$, the LTL realizability problem reduces to decide, given a tbUCW A over inputs Σ_2 and outputs Σ_1, whether there is a non-empty Moore machine M such that $L(M) \subseteq L_{\text{uc}}(A)$. If $L(M) \subseteq L_{\text{uc},K}(A)$ for some K, we say that the tbUKCW (A, K) is realizable.

Example 2 (running example) Fig. 1(a) represents a tbUCW equivalent to the formula $\Box(r \rightarrow \mathcal{X}(\Diamond g))$ [8], where r is an input signal and g is an output signal. States of Q_1 are denoted by circles while states of Q_2 are denoted by squares. State q_4 is denoted by a square because it is a final state. The transitions on missing letters are going to an additional sink non-final state that we do not represent for the sake of readability. If a request r is never granted, then a run will visit the final state q_4 infinitely often.

3 Safety Games

In this section, we provide a definition of safety games that is well-suited to support our synthesis methods detailed in the following sections. Player 1 will play the role of the system

[8] Note that this tbUCW is equivalent to the tbNBW of the negation of the specification, i.e., the formula $\Diamond(r \wedge \Box\neg g)$. So, tools for translation of LTL to NBW can be used to obtain the UCW when applied to the negation of the specification.



(a) UCW (b) Safety game $G(\phi, 1)$

Fig. 1 UCW and safety game for the formula $\phi \equiv \Box(r \rightarrow \mathcal{X}\Diamond g)$

while Player 2 will play the role of the environment. This is why, as the reader will see, our definition of games is asymmetric. In those games, Players move alternatively.

Turn-based games A *turn-based game* on a finite set of moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$ such that $\mathsf{Moves}_2 \neq \varnothing$ is a tuple $G = (S_1, S_2, \Gamma_1, \Delta_1, \Delta_2)$ where:

- (i) S_1 is the set of Player 1 positions, S_2 is the set of Player 2 positions, $S_1 \cap S_2 = \varnothing$. We let $S = S_1 \uplus S_2$.
- (ii) $\Gamma_1 : S_1 \rightarrow 2^{\mathsf{Moves}_1}$ is a function that assigns to each position of Player 1 the subset of moves that are available in that position. For Player 2, we assume that all the moves in Moves_2 are available in all the positions $s \in S_2$.
- (iii) $\Delta_1 : S_1 \times \mathsf{Moves}_1 \rightarrow S_2$ is a partial function that maps a pair (s, m) to the position reached from s when Player 1 chooses $m \in \Gamma_1(s)$. $\Delta_2 : S_2 \times \mathsf{Moves}_2 \rightarrow S_1$ is a function that maps (s, m) to the position reached from s when Player 2 chooses m.

We define the partial function Δ as the union of the partial function Δ_1 and the function Δ_2. Unless stated otherwise, we fix for the sequel of this section a turn-based game $G = (S_1, S_2, \Gamma_1, \Delta_1, \Delta_2)$ on moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$.

Given a function $\Lambda : S_1 \rightarrow 2^{\mathsf{Moves}_1}$, the *restriction of G by Λ* is the game $G[\Lambda] = (S_1, S_2, \widehat{\Gamma_1}, \widehat{\Delta_1}, \Delta_2)$ where for all $s \in S_1$, $\widehat{\Gamma_1}(s) = \Gamma_1(s) \cap \Lambda(s)$ and $\widehat{\Delta_1}$ equals Δ_1 on the domain restricted to the pairs $\{(s, m) \mid s \in S_1 \wedge m \in \widehat{\Gamma_1}(s)\}$, i.e., $G[\Lambda]$ is as G but with the moves of Player 1 restricted by Λ.

Rules of the game The game on G is played in rounds and generates a finite or an infinite sequence of positions that we call a *play*. In the initial round, the game is in some position, say s_0, and we assume that Player 1 owns that position. Then if $\Gamma_1(s_0)$ is non-empty Player 1 chooses a move $m_0 \in \Gamma_1(s_0)$, and the game evolves to position $s_1 = \Delta_1(s_0, m_0)$, otherwise the game stops. If the game does not stop then the next round starts in s_1. Player 2 chooses a move $m_1 \in \mathsf{Moves}_2$ and the game proceeds to position $s_2 = \Delta_2(s_1, m_1)$. The game proceeds accordingly either for an infinite number of rounds or it stops when a position

$s \in S_1$ is reached such that $\Gamma_1(s) = \varnothing$. Player 1 wins if the game does not stop otherwise Player 2 wins (safety winning condition). Our variant of safety games are thus zero-sum games as usual. In particular, the positions $s \in S_1$ such that $\Gamma_1(s) \neq \varnothing$ are the safe positions of Player 1.

Plays and strategies We now define formally the notions of play, strategy, outcome of a strategy and winning strategies. Given a sequence $\rho = s_0 s_1 \dots s_n \dots \in S^* \cup S^\omega$, we denote by $|\rho|$ its length (which is equal to ω if ρ is infinite). Given a non-empty sequence ρ, we denote by $\mathsf{first}(\rho)$ the first element of ρ, and if ρ is finite, we denote by $\mathsf{last}(\rho)$ its last element.

A *play* in G is a finite or infinite sequence of positions $\rho = s_0 s_1 \dots s_n \dots \in S^* \cup S^\omega$ such that : (i) if ρ is finite then $\mathsf{last}(\rho) \in S_1$ and $\Gamma_1(\mathsf{last}(\rho)) = \varnothing$; (ii) ρ is consistent with the moves and transitions of G, i.e., for all i, $0 \leq i \leq |\rho|$, we have that $s_{i+1} = \Delta(s_i, m)$ for some $m \in \Gamma_1(s_i)$ if $s \in S_1$, or $m \in \mathsf{Moves}_2$ if $s \in S_2$.We denote by $\mathsf{Plays}(G)$ the set of plays in G.

Given a set of finite or infinite sequences $L \subseteq S^* \cup S^\omega$, we write $\mathsf{Pref}_j(L)$, $j \in \{1, 2\}$, for the set of prefixes of sequences in L that end up in a position of Player j. Let $\perp$ be such that $\perp \notin \mathsf{Moves}$. A *strategy for Player 1* in G is a function $\lambda_1 : \mathsf{Pref}_1(\mathsf{Plays}(G)) \to \mathsf{Moves}_1 \cup \{\perp\}$ which is consistent with the set of available moves, i.e., for all $\rho \in \mathsf{Pref}_i(\mathsf{Plays}(G))$, we have that: (i) $\lambda_1(\rho) \in \Gamma_1(\mathsf{last}(\rho)) \cup \{\perp\}$, and (ii) $\lambda_1(\rho) = \perp$ only if $\Gamma_1(\mathsf{last}(\rho)) = \varnothing$. A *strategy for Player 2* in G is a function $\lambda_2 : \mathsf{Pref}_2(\mathsf{Plays}(G)) \to \mathsf{Moves}_2$. Note that a Player 2's strategy never contains $\perp$ as all the moves of Player 2 are allowed at any position, whereas the moves of Player 1 are restricted by Γ_1.

A play $\rho = s_0 s_1 \dots s_n \dots \in \mathsf{Plays}(G)$ is *compatible* with a strategy λ_j of Player j ($j \in \{1, 2\}$), if for all i, $0 \leq i < |\rho|$, if $s_i \in S_j$ then $s_{i+1} = \Delta_j(s_i, \lambda_j(s_0 s_1 \dots s_i))$.We denote by $\mathsf{outcome}(G, s, \lambda_j)$ the subset of plays in $\mathsf{Plays}(G)$ that are compatible with the strategy λ_j of Player j, and that start in s. We denote by $\mathsf{outcome}(G, s, \lambda_1, \lambda_2)$ the unique play that is compatible with both λ_1 and λ_2, and starts in s.

The *winning* plays for Player 1 are those that are infinite, i.e., $\mathsf{Win}_1(G) = \mathsf{Plays}(G) \cap S^\omega$, or equivalently those that never reach an unsafe position $s \in S_1$ of Player 1, i.e., a position s such that $\Gamma_1(s) = \varnothing$. A strategy λ_1 is *winning* in G from s_{ini} iff $\mathsf{outcome}(G, s_{\mathsf{ini}}, \lambda_1) \subseteq \mathsf{Win}_1(G)$ or equivalently, if the plays compatible with λ_1 are infinite. We call a turn-based game with such a winning condition in mind a *safety game* . We denote by $\mathsf{WinPos}_1(G)$ the subset of positions $s \in S$ in G for which there exists λ_1 such that $\mathsf{outcome}(G, s, \lambda_1) \subseteq \mathsf{Win}_1(G)$.

Games with initial position A *safety game with initial position* is a pair (G, s_{ini}) where $s_{\mathsf{ini}} \in S_1 \cup S_2$ is a position of the game structure G called the *initial position*. The set of plays in (G, s_{ini}) are the plays of G starting in s_{ini}, i.e., $\mathsf{Plays}(G, s_{\mathsf{ini}}) = \mathsf{Plays}(G) \cap s_{\mathsf{ini}} \cdot (S^* \cup S^\omega)$. All the previous notions carry over to games with initial positions.

We now recall two classical algorithms to solve safety games. One explores the game backward while the other explores it in a forward fashion.

Backward algorithm for solving safety games The classical fixpoint algorithm to solve safety games relies on iterating the following monotone operator over sets of game positions, see [12] for example. The safety games we defined alternate between positions of Player 1 and positions of Player 2. We define two operators $\mathsf{Pre}_1 : 2^{S_2} \to 2^{S_1}$ and $\mathsf{Pre}_2 : 2^{S_1} \to 2^{S_2}$ such that for all $X_1 \subseteq S_1$, $\mathsf{Pre}_2(X_1)$ are the positions from which Player 2 cannot avoid

reaching X_1 in one step. For all $X_2 \subseteq S_2$, $\mathsf{Pre}_1(X_2)$ are the positions from which Player 1 can reach S_2 in one step. We also define $\mathsf{CPre} : 2^{S_1} \to 2^{S_1}$ as $\mathsf{Pre}_1 \circ \mathsf{Pre}_2$:

$$\begin{array}{ll} \mathsf{Pre}_1(X_2) & = \{s \in S_1 \mid \exists m \in \Gamma_1(s), \Delta_1(s,m) \in X_2\} \\ \mathsf{Pre}_2(X_1) & = \{s \in S_2 \mid \forall m \in \mathsf{Moves}_2, \Delta_2(s,m) \in X_1\} \\ \mathsf{CPre} & = \mathsf{Pre}_1 \circ \mathsf{Pre}_2 \end{array}$$

Now, we define the following sequence of subsets of positions:

$$W_0 = \{s \in S_1 \mid \Gamma_1(s) \neq \varnothing\} \qquad W_i = W_{i-1} \cap \mathsf{CPre}(W_{i-1}) \text{ for all } i \geq 1$$

Denote by $W^\natural$ the fixpoint of this sequence. It is well known that $W^\natural = \mathsf{WinPos}_1(G)$.

Forward algorithm for solving safety games We describe an algorithm that computes the winning positions in a safety game in a forward fashion, starting from the initial position of the game. The algorithm explores the positions of the game and once a position is known to be losing, this information is back propagated to the predecessors. A position of Player 1 is losing (for Player 1) iff it has no successors or all its successors are losing. A position of Player 2 is losing (for Player 1) iff one of its successors is losing. For solving safety games, we use an variant of OTFUR algorithm of [5] (Algo 1) based on an algorithm of [20]. At each step, the algorithm maintains an under-approximation Losing of the set of losing positions. The algorithm has a waiting-list $Waiting$ for reachable position exploration and reevaluation of positions. In particular, an edge is put in the waiting-list if it is the first time it has been reached, or the status of its target position has changed. The latter case means that when the information that a position is losing is known, this information is back-propagated to all its predecessors. A set $Passed$ records the visited positions. Finally, a set $Depend$ stores the edges (s, s') which need to be reevaluated when the value of s' changes.

At each step, the algorithm picks an edge $e = (s, s')$ in the waiting list. If its target s' has never been visited, one checks whether this target is obviously losing (when it has no successors). In this case, we add e in the waiting list for reevaluation. This amounts to back propagate the information on s'. Otherwise we add all the successors of s' in the waiting list for reevaluation. If s' has already been visited, then we compute the value of s. If s is losing, this information is back propagated to the positions whose safeness depends on s.

The overall complexity of this algorithm is linear in the size of the game, as every edge is added at most twice to the waiting list. We refer the reader to [20, 5] for the formal proof of its correctness, expressed by the following theorem:

Theorem 1 *After termination of Algorithm 1, the set of positions s such that $\neg\mathsf{Losing}[s]$ is equal to the winning positions (for Player 1) reachable from the initial position.*

4 From UCW to U*K*CW Realizability

In this section, we reduce the realizability problem with a specification given by a turn-based universal co-Büchi automaton (tbUCW) to a specification given by a turn-based universal K-co-Büchi automaton (tbU*K*CW). A variant of this lemma expressed on universal co-Büchi tree automata has been proved in [28].

Lemma 1 *Let A be a* tbUCW *over inputs Σ_2 and outputs Σ_1 with n states, and M be a Moore machine over inputs Σ_2 and outputs Σ_1 with m states. Then $L(M) \subseteq L_{uc}(A)$ iff $L(M) \subseteq L_{uc,2mn}(A)$.*

Algorithm 1: OTFUR [5] algorithm for safety games

Data: G, s_{ini}
//Initialization
1 Passed $:= \{s_{\mathsf{ini}}\}$; Depend$(s_{\mathsf{ini}}) := \varnothing$;
2 **for** all position s **do** Losing$[s]$:= false;
3 Waiting $:= \{(s_{\mathsf{ini}}, s') | \exists m \in \Gamma_1(s_{\mathsf{ini}}) : s' = \Delta_1(s_{\mathsf{ini}}, m)\}$;
//Saturation
4 **while** Waiting $\neq \varnothing \wedge \neg$Losing$[s_{\mathsf{ini}}]$ **do**
5 $e = (s, s') := pop$(Waiting);
6 **if** $s' \notin$ Passed **then**
7 Passed := Passed $\cup \{s'\}$;
8 Losing$[s']$:= $s' \in S_1 \wedge \Gamma_1(s') = \varnothing$;
9 Depend$[s']$:= $\{(s, s')\}$;
10 **if** Losing$[s']$ **then**
11 Waiting := Waiting $\cup \{e\}$; //add e for reevaluation
12 **else**
13 Waiting := Waiting $\cup \{(s', s'') | \exists m \in \Delta(s') : s'' = \Delta(s', m)\}$;
14 **else**
//reevaluation
Losing* := $s \in S_1 \wedge \bigwedge_{m \in \Gamma_1(s), s'' = \Delta_1(s,m)}$ Losing$[s'']$
15 $\vee\ s \in S_2 \wedge \bigvee_{m \in \mathsf{Moves}_2, s'' = \Delta_2(s,m)}$ Losing$[s'']$;
16 **if** Losing* **then**
17 Losing$[s]$:= true;
18 Waiting := Waiting $\cup$ Depend$[s]$ //back propagation
19 **if** $\neg$Losing$[s']$ **then** Depend$[s']$:= Depend$[s'] \cup \{e\}$
20 **return** $\neg$Losing$[s_{\mathsf{ini}}]$

Proof The back direction is obvious since $L_{\mathrm{uc},k}(A) \subseteq L_{\mathrm{uc}}(A)$ for all $k \in \mathbb{N}$. We sketch the forth direction. Informally, the infinite paths of M starting from the initial state define words that are accepted by A. Therefore in the product of M and A, there is no cycle visiting a final state of A, which allows one to bound the number of visited final states by the number of states in the product. More formally, we first transform M into a tbNBW A_M such that $L(M) = L_{\mathsf{b}}(A_M)$. It suffices to copy every state of M and to define the transitions as follows: if $q \xrightarrow{i} q'$ is a transition of M with $i \in \Sigma_2$, and the output of q is $o \in \Sigma_1$, then we transform this transition into the two A_M transitions $q \xrightarrow{o} q^c$ and $q^c \xrightarrow{i} q'$ where q^c is a fresh state denoting a copy of q. All states of A_M are set to be final, so that $L_{\mathsf{b}}(A_M)$ is exactly the set of traces of infinite paths of A_M (viewed as an edge-labeled graph) starting from the initial state. By hypothesis, $L_{\mathsf{b}}(A_M) \subseteq L_{\mathrm{uc}}(A)$. Note that A_M has $2m$ states.

Let Q be the set of states of A, Q_{A_M} the set of states of A_M and $A \times A_M$ the product of A and A_M, i.e. the automaton over $\Sigma_2 \cup \Sigma_1$ whose set of states is $Q \times Q_{A_M}$, initial states are pairs of initial states, and transitions have the form $(q, p) \xrightarrow{\sigma} (q', p')$ for all transitions $q \xrightarrow{\sigma} q'$ of A and $p \xrightarrow{\sigma} p'$ of A_M. Since $L_{\mathsf{b}}(A_M) \subseteq L_{\mathrm{uc}}(A)$, there is no cycle in $A \times A_M$ reachable from an initial state and that contains a state (q, p) where $q \in Q$ is final. Indeed, otherwise there would exist an infinite path in $A \times A_M$, visiting (q, p) infinitely often. Every infinite word obtained as a trace of this path would be accepted by A_M but not by A (since there would be a run on it visiting q infinitely often). Therefore the runs of A on words accepted by A_M visit at most $2nm$ final states, where n (resp. $2m$) is the number of states of A (resp. A_M). □

The following result is proved in Th. 4.3 of [19], as a small model property of universal co-Büchi tree automata. We also prove it here for the sake of self-containedness.

Lemma 2 *Given a realizable* tbUCW A *over inputs* Σ_2 *and outputs* Σ_1 *with* n *states, there exists a non-empty Moore machine with at most* $n^{2n+2}+1$ *states that realizes it.*

Proof We first sketch the proof. In the first step, we show by using Safra's determinization of NBWs that A is equivalent to a turn-based deterministic and complete parity automaton A^d. Using a result from [22], we know that A^d has at most $m := 2n^{2n+2}+2$ states. We then view A^d has a turn-based two-player parity game $G(A^d)$ (with at most m states) such that A^d (or equivalently A) is realizable iff Player 1 has a winning strategy in $G(A^d)$. It is known that parity games admit memoryless strategies [12]. Therefore if A^d is realizable, there exists a strategy for Player 1 in $G(A^d)$ that can be obtained by removing all but one outgoing edge per Player 1's state. We can finally transform this strategy into a Moore machine with at most $n^{2n+2}+1$ states that realizes A^d (and A).

More formally, the proof uses the parity acceptance condition for automata and for games. Given an automaton B with state set Q_B, a parity acceptance condition is given by a mapping c from Q_B to $\mathbb{N}$. A run ρ is accepting if $min\{c(q) \mid q \in \mathsf{Inf}(\rho)\}$ is even, where $\mathsf{Inf}(\rho)$ is the set of $q \in Q$ that appear infinitely many times along ρ. Final states are not needed in B for this acceptance condition. We denote by $L_{par,c}(B)$ the language accepted by B under the parity acceptance condition c.

Given a turn-based two-player game $G = (S_1, S_2, s_0, \Delta)$, the parity winning condition is given by a mapping $c : S_1 \cup S_2 \to \mathbb{N}$. In that case, for all $i \in \{1,2\}$, a strategy λ_i for Player i is winning if $\mathsf{Outcome}_G(\lambda_i) \subseteq \{\pi \in (S_1S_2)^\omega \mid min\{c(s) \mid s \in \mathsf{Inf}(\pi)\} \text{ is even}\}$.

Let $A = (\Sigma_1, \Sigma_2, Q_1, Q_2, q_0, \alpha, \delta_1, \delta_2)$. We let $Q = Q_1 \cup Q_2$ and $\delta = \delta_1 \cup \delta_2$. Let A_{OI} be the automaton $(\Sigma, Q, q_0, \alpha, \delta)$. We denote by $m : (\Sigma_1\Sigma_2)^\omega \to \Sigma^\omega$ the function that maps any word $w = o_0 i_0 o_1 i_1 \dots$ to $m(w) = (o_0 \cup i_0)(o_1 \cup i_1)\dots$. Note that m admits an inverse denoted by m^{-1}. We have that $m(L_{\mathsf{b}}(A_{OI})) = L_{\mathsf{b}}(A)$ (*). By Safra's determinization, there exists a deterministic parity automaton D_{OI} with a parity condition c such that $L_{par,c}(D_{OI}) = L_{\mathsf{b}}(A_{OI})$. Moreover, by [22], we can assume that D_{OI} has at most n^{2n+2} states. Since $L_{par,c}(D_{OI}) \subseteq (\Sigma_2\Sigma_1)^\omega$, it is easy to transform D_{OI} into a deterministic turn-based parity automaton D with a parity condition c' such that $L_{par,c}(D_{OI}) = m^{-1}(L_{par,c'}(D))$: it suffices to take the product with the two-states automaton that accepts $(\Sigma_1\Sigma_2)^\omega$ (let i and o its two states). The states of the product are therefore pairs (q,p) with q a state of D_{OI} and $p \in \{i,o\}$, and we let $c'(q,p) = c(q)$. Note that D has at most $2n^{2n+2}$ states. From equality (*) and the equalities $L_{par,c}(D_{OI}) = m^{-1}(L_{par,c'}(D))$ and $L_{par,c}(D_{OI}) = L_{\mathsf{b}}(A_{OI})$, we get $L_{par,c'}(D) = L_{\mathsf{b}}(A)$. Then we complete the automaton D by adding two dead states and get a complete deterministic turn-based automaton A^d (with at most $2n^{2n+2}+2$ states). Finally, we take the dual parity condition $c_d = c'+1$ which increments the value of each state by 1, so that $L_{par,c_d}(A^d) = \Sigma^\omega - L_{par,c'}(D) = \Sigma^\omega - L_{\mathsf{b}}(A)$, from which we get $L_{\mathsf{uc}}(A) = L_{par,c_d}(A^d)$.

Let $A^d = (\Sigma_1, \Sigma_2, Q^d_I, Q^d_O, q_0, \delta^d_2, \delta^d_1)$ and $Q^d = Q^d_I \cup Q^d_O$. We now view A^d has a turn-based two-player parity game $G(A^d) = (Q^d_0, Q^d_2, q_0, \Delta)$: Q^d_1 are Player 1's states (q_0 being the initial state) while Q^d_2 are Player 2's states, and we put a transition $(q,p) \in \Delta$ from a state $q \in Q^d$ to a state $p \in Q^d$ if there exists $\sigma \in \Sigma_2 \cup \Sigma_1$ and a transition $q \xrightarrow{\sigma} p$ in A^d. Since A^d has at most $2n^{2n+2}+2$ states, $G(A^d)$ has also at most $2n^{2n+2}+2$ states.

The specification A^d is realizable (or equivalently A is realizable) iff Player 1 has a winning strategy in $G(A^d)$. Therefore if A is realizable, Player 1 has a winning strategy in $G(A^d)$ given by a mapping γ from Q^d_1 to Q^d_2 such that $\mathsf{Outcome}_{G(A^d)}(\gamma)$ are words ρ over

$(Q_1^d Q_2^d)^\omega$ such that $min\{c(q) \mid q \in \mathsf{Inf}(\rho)\}$ is even. Moreover, those words correspond to accepting runs of A^d on words over Σ. Therefore the strategy γ can easily be used to define a Moore machine M such that $L(M) \subseteq L_{par,c}(A^d) = L_{\text{uc}}(A)$: first we assume that Σ is totally ordered. The machine M is defined as follows: Q_1^d are its states, q_0 is the initial state, the output function g is defined by $g(q) = min\{\sigma_o \mid (q, \sigma_o, \gamma(q)) \in \delta_1^d\}$, for all $q \in Q_1^d$, and finally we put a transition $q \xrightarrow{\sigma_i} q'$, for all $q, q' \in Q_1^d$, and all $\sigma_i \in \Sigma_2$ if $\gamma(q) \xrightarrow{\sigma_i} q' \in \delta_2^d$. Note that the transition relation of M is a (total) function since A^d is complete, and has less than $(2n^{2n+2} + 2)/2 = n^{2n+2} + 1$ states. □

The following theorem states that we can reduce the realizability of a tbUCW specification to the realizability of a tbU*K*CW specification.

Theorem 2 *Let A be a* tbUCW *over Σ_2, Σ_1 with n states and $K = 2n(n^{2n+2} + 1)$. Then A is realizable iff (A, K) is realizable.*

Proof If A is realizable, by Lem. 2, there is a non-empty Moore machine M with m states ($m \leq n^{2n+2} + 1$) realizing A. Thus $L(M) \subseteq L_{\text{uc}}(A)$ and by Lem. 1, it is equivalent to $L(M) \subseteq L_{\text{uc},2mn}(A)$. We can conclude since $L_{\text{uc},2mn}(A) \subseteq L_{\text{uc},K}(A)$ ($2mn \leq K$). The converse is obvious as $L_{\text{uc},K}(A) \subseteq L_{\text{uc}}(A)$. □

5 From U*K*CW Realizability to Safety Games

In the last section, we reduced the tbUCW realizability problem to the tbU*K*CW realizability problem. In this section, we reduce this new problem to a safety game. It is based on the determinization of tbU*K*CWs into complete turn-based deterministic 0-Co-Büchi automata, which can obviously be viewed as safety games.

Determinization of U*K*CW Let A be a tbU*K*CW $(\Sigma_1, \Sigma_2, Q_1, Q_2, q_0, \alpha, \Delta_1, \Delta_2)$ with $K \in \mathbb{N}$. We let $Q = Q_1 \cup Q_2$ and $\Delta = \Delta_1 \cup \Delta_2$. It is easy to construct an equivalent complete turn-based deterministic 0-co-Büchi automaton $\det(A, K)$. Intuitively, it suffices to extend the usual subset construction with counters, for all $q \in Q$, that count (up to $K+1$) the maximal number of accepting states which have been visited by runs ending up in q. We set the counter of a state q to -1 when no run on the prefix read so far ends up in q. The final states are the sets in which a state has its counter greater than K. For any $n \in \mathbb{N}$, $[n]$ denotes the set $\{-1, 0, 1, \ldots, n\}$. Formally, we let $\det(A, K) = (\Sigma_1, \Sigma_2, \mathcal{F}_1, \mathcal{F}_2, F_0, \alpha', \delta_1, \delta_2)$ where:

$$
\begin{array}{lll}
\mathcal{F}_1 & = \{F \mid F \text{ is a mapping from } Q_1 \text{ to } [K+1]\} & \\
\mathcal{F}_2 & = \{F \mid F \text{ is a mapping from } Q_2 \text{ to } [K+1]\} & \\
F_0 & = q \in Q_1 \mapsto \begin{cases} -1 & \text{if } q \neq q_0 \\ (q_0 \in \alpha) & \text{otherwise} \end{cases} & \\
\alpha' & = \{F \in \mathcal{F}_2 \cup \mathcal{F}_1 \mid \exists q, F(q) > K\} & \\
\mathsf{succ}(F, \sigma) & = q \mapsto \max\{\min(K+1, F(p) + (q \in \alpha)) \mid q \in \Delta(p, \sigma), F(p) \neq -1\} & \\
\delta_1 & = \mathsf{succ}|_{\mathcal{F}_1 \times \Sigma_1} & \delta_2 = \mathsf{succ}|_{\mathcal{F}_2 \times \Sigma_2}
\end{array}
$$

where $\max \emptyset = -1$, and $(q \in \alpha) = 1$ if q is in α, and 0 otherwise. The automaton $\det(A, K)$ has the following properties:

Proposition 1 *Let A be a* tbUCW *and $K \in \mathbb{N}$. Then $\det(A, K)$ is deterministic, complete, and $L_{uc,0}(\det(A, K)) = L_{uc,K}(A)$.*

Reduction to a safety game Finally, we define the game $G(A,K)$ as follows: it is $\det(A,K)$ where input states are viewed as Player 2's states and output states as Player 1's states. Formally, we define $G(A,K) = (\mathcal{F}_1, \mathcal{F}_2, \Gamma_1, \Delta_1, \Delta_2, F_0)$ over the set of moves $\mathsf{Moves}_1 = \Sigma_1$ and $\mathsf{Moves}_2 = \Sigma_2$, where F_0 is the initial position. The set of available moves in a Player 1's position are defined via a successor function succ. An action $\sigma_1 \in \mathsf{Moves}_1$ is available for Player 1 in a position $F \in S_1$ if the counters of F and $\mathsf{succ}(F,\sigma_1)$ do not exceed K. More formally, $\sigma_1 \in \Gamma_1(F)$ iff for all $p \in Q_1$ and all $q \in Q_2$, $F(p) \leq K$ and $\mathsf{succ}(F,\sigma_1)(q) \leq K$. The transition function Δ_1 is defined by $\Delta_1(F,\sigma) = \mathsf{succ}(F,\sigma)$ for all $F \in S_1$ and all $\sigma \in \Gamma_1(s)$. The function Δ_2 is defined by $\Delta_2(F,\sigma_2) = \mathsf{succ}(F,\sigma_2)$ for all $F \in S_2$ and all $\sigma_2 \in \mathsf{Moves}_2$.

As an obvious consequence of Th. 2 and Prop. 1, we get:

Theorem 3 (Reduction to a safety game[9]) *Let A be a* tbUCW *over inputs Σ_2 and outputs Σ_1 with n states ($n > 0$), and let $K = 2n(n^{2n+2}+1)$. The specification A is realizable iff Player 1 has a winning strategy in the game $G(A,K)$.*

Proof Suppose that A is realizable. By Theorem 2 and Proposition 1, (A,K) is also realizable, as well as $\det(A,K)$. Thus there is exists a non-empty Moore machine M over inputs Σ_2 and outputs Σ_1 such that $L(M) \subseteq L_{\text{uc},0}(\det(A,K))$. We now construct a winning strategy γ for Player 1 in $G(A,K)$. Intuitively, $\mathsf{Outcome}_{G(A,K)}(\gamma)$ will correspond to runs of $\det(A,K)$ on words of $L(M)$. Therefore, since $L(M) \subseteq L_{\text{uc},0}(\det(A,K))$, $\mathsf{Outcome}_{G(A,K)}(\gamma)$ won't visit final states. For the sake of clarity, we view this Moore machine as a (total) mapping $\lambda : \Sigma_2^* \to \Sigma_1$. First assume that Σ_1 and Σ_2 are totally ordered by some order $\prec$. We define a strategy γ in $G(A,K)$ inductively on its outcome. First, $\gamma(F_0) = \lambda(\epsilon)$, where ϵ is the empty sequence. Clearly, since λ is winning, $\lambda(\epsilon)$ is an available move in F_0. Then, for any finite outcome $H = F_0^1 F_0^2 \dots F_{m-1}^2 F_m^1 \in (\mathcal{F}_1\mathcal{F}_2)^*\mathcal{F}_1$ of γ of length m, we associate H with the word $w(H)$ defined as follows: $w(H) = \sigma_1^I \dots \sigma_m^I$ where for all $1 \leq i \leq m$, $\sigma_i^I = min\{\sigma \mid \mathsf{succ}(F_{i-1}^2, \sigma) = F_i^1\}$. We let $\gamma(H) = \lambda(w(H))$. It is easy to prove by induction that $\lambda(w(H))$ is an available move at position F_m^1 (this is because λ is winning and generates outcomes that are accepted by $\det(A,K)$). The strategy λ is clearly winning since it always chooses moves that are available, so that its outcomes are all infinite.

Conversely, suppose that Player 1 has a winning strategy γ in $G(A,K)$. It is known that we can assume that γ is memoryless [12]. So let γ be a mapping from $\mathcal{F}_1$ to Σ_1. We construct a winning strategy for the controller, represented as a Moore machine $M_\gamma = (\Sigma_1, \Sigma_2, \mathcal{F}_1, F_0, \delta_\gamma, \gamma)$. Its state set is $\mathcal{F}_1$ with initial state F_0; for all $F \in \mathcal{F}_1$, the output of F is $\gamma(F)$; for all $F \in \mathcal{F}_1$, and all $\sigma_i \in \Sigma_2$, the transition function is defined by $\delta_\gamma(F,\sigma_i) = \delta_2(\delta_1(F,\gamma(F)),\sigma_i)$. Since γ is winning, it is clear by construction that all states of M_γ reachable from the initial state are non-final. Therefore $L(M_\gamma) \subseteq L_{\text{uc},0}(\det(A,K)) = L_{\text{uc}}(A)$. Moreover, the transition relation of M_γ is a (total) function, as $\det(A,K)$ is complete. Since there is a winning strategy γ in $G(A,K)$, it means that $G(A,K)$ is non-empty, and so is M_γ, which concludes the proof. □

Associating a safety game with an LTL formula ϕ is done as follows: (1) construct a UCW A_ϕ equivalent to ϕ, (2) construct $G(A_\phi,K)$, denoted as $G(\psi,K)$ in the sequel, where $K = 2n(n^{2n+2}+1)$ and n is the number of states of A_ϕ.

[9] An similar result expressed on co-Büchi tree automata can be found in [28].

Incremental Algorithm. In practice, for checking the existence of a winning strategy for Player 1 in the safety game, we rely on an incremental approach. We use the following property of UK CWs: for all $k_1, k_2 \cdot 0 \leq k_1 \leq k_2 \cdot L_{uc,k_1}(A) \subseteq L_{uc,k_2}(A) \subseteq L_{uc}(A)$. So, the following theorem which is a direct consequence of the previous property allows us to test the existence of strategies for increasing values of K:

Theorem 4 *For all* tbUCWs A*, for all* $k \geq 0$*, if Player* 1 *has a winning strategy in the game* $G(A, k)$ *then the specification defined by* A *is realizable.*

Example 3 Fig. 1(b) represents the safety game $G(\phi, 1)$ where $\phi = \Box(r \rightarrow \mathcal{X}\Diamond g)$. Positions are pairs of states of the UCW with their counter values. Player 1's positions are denoted by circles while Player 2's positions are denoted by squares. The unavailable move of Player 1 from position $(q_2, 0)$ is denoted by a dashed arrow. It goes to a position where a counter exceeds the value K. Any winning strategy in this game is a strategy which chooses the moves attached to plain arrows, as indeed Player 1 wins the game iff she never follows the dashed arrow.

Unrealizable Specifications. The incremental algorithm is not reasonable to test unrealizability. Indeed, with this algorithm it is necessary to reach the bound $2n(n^{2n+2} + 1)$ to conclude for unrealizability. To obtain a more practical algorithm, we rely on the determinacy of ω-regular games (a corollary of the general result by Martin [21]).

Theorem 5 *For all* LTL *formulas* ϕ*, either* (i) *there exists a Player* 1*'s strategy* λ_1 *s.t. for all Player* 2*'s strategies* λ_2*,* $\mathsf{outcome}(\lambda_1, \lambda_2) \models \phi$*, or there exists a Player* 2*'s strategy* λ_2 *s.t. for all Player* 1*'s strategies* λ_1*,* $\mathsf{outcome}(\lambda_1, \lambda_2) \models \neg\phi$*.*

So, when an LTL specification ϕ is not realizable for Player 1, it means that $\neg\phi$ is realizable for Player 2. To avoid in practice the enumeration of values for K up to $2n(n^{2n+2} + 1)$, we propose the following algorithm. First, given the LTL formula ϕ, we construct two UCWs: one that accepts all the models of ϕ, denoted by A_ϕ, and one that accepts all the models of $\neg\phi$, denoted by $A_{\neg\phi}$. Then we check realizability by Player 1 of ϕ, and in parallel realizability by Player 2 of $\neg\phi$, incrementing the value of K. When one of the two processes stops, we know if ϕ is realizable or not. In practice, we will see that either ϕ is realizable for Player 1 for a small value of K or $\neg\phi$ is realizable for Player 2 for a small value of K. In the next section, we show that the game $G(A, K)$ has a nice structure that allows to compactly represent and efficiently manipulate sets of winning positions.

6 Antichain-based Symbolic Algorithms

In the previous section, we have shown how to reduce the realizability problem of a UCW A with n states to a family of safety game $G(A, k)$ for $0 \leq k \leq 2n(n^{2n+2} + 1)$. From now on, we fix some $k \in \mathbb{N}$. In this section, we show that the positions of the game $G(A, k)$ can be partially ordered. We also show that the sets of positions manipulated during the backward algorithm for safety games (see Section 3) are downward closed for this order. This allows to compactly represent those sets by the antichain of their maximal elements. Therefore it is not necessary to construct explicitly the game $G(A, k)$, which may be very large even for small values of k. We also show that the forward algorithm for safety games can also benefit from the particular structure of $G(A, k)$.

Recall that $G(A,k)$ is defined by the tuple $(\mathcal{F}_1,\mathcal{F}_2,\Gamma_1,\Delta_1,\Delta_2,F_0)$ over the set of moves Σ_1 for Player 1 (controller) and Σ_2 for Player 2 (environment). We also let $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$, and we denote by $\mathsf{safe} \subseteq \mathcal{F}$ the set of counting functions whose counters do not exceed k.

Ordering of game positions We define the relation $\preceq \subseteq \mathcal{F}_2 \times \mathcal{F}_2 \cup \mathcal{F}_1 \times \mathcal{F}_1$ by

$$F \preceq F' \text{ iff } \forall q,\ F(q) \leq F'(q).$$

It is clear that $\preceq$ is a partial order. Intuitively, if Player 1 can win from F' then she can also win from all $F \preceq F'$, as it is somehow more difficult to stay below the bound k from F' than from F. Formally, $\preceq$ is a game simulation relation in the terminology of [2].

Closed sets and antichains. A set $S \subseteq \mathcal{F}$ is *closed for* $\preceq$, if $\forall F \in S \cdot \forall F' \preceq F \cdot F' \in S$. We usually omit references to $\preceq$ if clear from the context. Let S_1 and S_2 be two closed sets, then $S_1 \cap S_2$ and $S_1 \cup S_2$ are closed. The *closure* of a set $S \subseteq \mathcal{F}$, denoted by $\downarrow S$, is the set $S' = \{F' \in \mathcal{F} \mid \exists F \in S \cdot F' \preceq F\}$. Note that for all closed sets $S \subseteq \mathcal{F}$, $\downarrow S = S$. A set $L \subseteq \mathcal{F}$ is an *antichain* if all elements of L are incomparable for $\preceq$. Let $S \subseteq \mathcal{F}$, we denote by $\lceil S \rceil$ the set of maximal elements of S, that is $\lceil S \rceil = \{F \in S \mid \nexists F' \in S \cdot F' \neq F \wedge F \preceq F'\}$, it is an antichain. If S is closed then $\downarrow\lceil S \rceil = S$, i.e. antichains are *canonical representations* for closed sets. Similarly, we denote by $\lfloor S \rfloor$ and $\uparrow S$ the minimal elements of S and its upward closure, respectively. Since the size of a state $F \in \mathcal{F}$ is in practice much smaller than the number of elements in the antichains, we consider that comparing two states is in constant time.

Proposition 2 *Let $L_1, L_2 \subseteq \mathcal{F}$ be two antichains and $F \in \mathcal{F}$, then:*

(i) $\downarrow L_1 \cup \downarrow L_2 = \downarrow\lceil L_1 \cup L_2 \rceil$, this antichain can be computed in time $O((|L_1|+|L_2|)^2)$ and its size is bounded by $|L_1|+|L_2|$;

(ii) $\downarrow L_1 \cap \downarrow L_2 = \downarrow\lceil L_1 \sqcap L_2 \rceil$, where $F_1 \sqcap F_2 : q \mapsto \min(F_1(q), F_2(q))$, this antichain can be computed in time $O(|L_1|^2 \times |L_2|^2)$ and its size is bounded by $|L_1| \times |L_2|$,

(iii) $\downarrow L_1 \subseteq \downarrow L_2$ iff $\forall F_1 \in L_1 \cdot \exists F_2 \in L_2 \cdot F_1 \preceq F_2$, which can be established in time $O(|L_1| \times |L_2|)$,

(iv) $F \in \downarrow L_1$ can be established in time $O(|L_1|)$.

6.1 Backward algorithm for safety games with antichains

The image of a closed set S by the functions Pre_2, Pre_1, and CPre are closed sets:

Lemma 3 *For all closed sets $S_1, S_2 \subseteq \mathcal{F}_2$, $S_3 \subseteq \mathcal{F}_1$, the sets $\mathsf{Pre}_1(S_1)$, $\mathsf{CPre}(S_2)$, and $\mathsf{Pre}_2(S_3)$ are closed.*

As a consequence, all the sets manipulated by the backward fixpoint algorithm are closed sets, and can therefore be compactly represented by the antichain of their maximal elements. Next, we show how to manipulate those sets efficiently.

Let us now turn to the computation of controllable predecessors. Let $F \in \mathcal{F}$, and $\sigma \in \Sigma_2 \cup \Sigma_1$. We denote by $\Omega(F,\sigma) \in \mathcal{F}$ the function defined by:

$$\Omega(F,\sigma) : q \in Q \mapsto \min\{\max(-1, F(q') - (q' \in \alpha)) \mid (q,\sigma,q') \in \delta\}$$

Note that since A is complete, the argument of min is a non-empty set. The function Ω is not the inverse of the function succ, as succ has no inverse in general. Indeed, it might be the case that a state $F \in \mathcal{F}$ has no predecessors or has more than one predecessor H such that $\mathsf{succ}(H, \sigma) = F$. However, we prove the following:

Proposition 3 *For all $F, F' \in \mathcal{F} \cap \mathsf{safe}$, and all $\sigma \in \Sigma_2 \cup \Sigma_1$,*

$$(i)\ F \preceq F' \implies \Omega(F,\sigma) \preceq \Omega(F',\sigma) \qquad (iii)\ F \preceq \Omega(\mathsf{succ}(F,\sigma),\sigma)$$
$$(ii)\ F \preceq F' \implies \mathsf{succ}(F,\sigma) \preceq \mathsf{succ}(F',\sigma) \qquad (iv)\ \mathsf{succ}(\Omega(F,\sigma),\sigma) \preceq F$$

Proof We establish the four items as follows:

(i) It holds as $\max(-1, F(q) - q \in \alpha) \leq \max(-1, F'(q) - q \in \alpha), \forall q \in Q$.

(ii) it holds as $\min(K+1, F(q') + q \in \alpha) \leq \min(K+1, F'(q') + q \in \alpha), \forall q' \in Q$.

(iii) Let $q \in Q$. We show that for all $q' \in \delta(q,\sigma)$, $F(q) \leq \mathsf{succ}(F,\sigma)(q') - (q' \in \alpha)$. This will be sufficient to conclude since it implies that $F(q) \leq max(-1, \mathsf{succ}(F,\sigma)(q') - (q' \in \alpha))$, for all $q' \in \delta(q,\sigma)$, and therefore that $F(q) \leq \Omega(\mathsf{succ}(F,\sigma),\sigma)(q)$. So let $q' \in \delta(q,\sigma)$, and let $I(q') = \{q'' \mid (q'',\sigma,q') \in \delta, F(q'') \neq -1\}$. Since $(q,\sigma,q') \in \delta$, we have $q \in I(q')$. We know that $\mathsf{succ}(F,\sigma)(q') = max\{min(K+1, F(q'') + q' \in \alpha) \mid q'' \in I(q')\}$. Since $q \in I(q')$, $\mathsf{succ}(F,\sigma)(q') \geq min(K+1, F(q) + q' \in \alpha)$. If $F(q) + q' \in \alpha \leq K+1$, then $\mathsf{succ}(F,\sigma)(q') - (q' \in \alpha) \geq F(q)$. The case $F(q) + (q' \in \alpha)$ is impossible since $F(q) \leq K$, as $F \in \mathsf{safe}$.

(iv) Let $q \in Q$. We first show that for all q' such that $(q',\sigma,q) \in \delta$ and $\Omega(F,\sigma)(q') \neq -1$, $\Omega(F,\sigma)(q') \leq F(q) - (q \in \alpha)$. This will be sufficient to conclude since it implies that $min(K+1, \Omega(F,\sigma)(q') + (q \in \alpha)) \leq F(q)$, for all q' such that $(q',\sigma,q) \in \delta$, and therefore that $\mathsf{succ}(\Omega(F,\sigma),\sigma)(q) \leq F(q)$. So let q' such that $(q',\sigma,q) \in \delta$ and $\Omega(F,\sigma)(q') \neq -1$. Let $I(q') = \{q'' \mid (q',\sigma,q'') \in \delta\}$. Since $(q',\sigma,q) \in \delta$, we have $q \in I(q')$. We know that $\Omega(F,\sigma)(q') = min\{max(-1\ ,\ F(q'') - (q'' \in \alpha)) \mid q'' \in I(q')\}$. Since $q \in I(q')$, we get $\Omega(F,\sigma)(q') \leq max(-1, F(q) - (q \in \alpha))$. The case $F(q) - (q \in \alpha) < -1$ is impossible, since otherwise we would have $\Omega(F,\sigma)(q') = -1$, which contradicts the hypothesis. Therefore $max(-1, F(q) - (q \in \alpha)) = F(q) - (q \in \alpha)$ and $\Omega(F,\sigma)(q') \leq F(q) - (q \in \alpha)$. □

Example 4 We illustrate point (iii) of the proposition 3 on the example of Fig. 1(a). Let us consider the position $[q_3 \mapsto 1]$. While $\mathsf{succ}([q_2 \mapsto 0, q_4 \mapsto 1], r) = [q_3 \mapsto 1]$, we have that $\Omega([q_3 \mapsto 1], r) = [q_2 \mapsto 1, q_4 \mapsto 1]$, i.e., $\Omega([q_3 \mapsto 1], r)$ returns the largest possible predecessor of $[q_3 \mapsto 1]$ by r, so $\mathsf{succ}([q_2 \mapsto 0, q_4 \mapsto 1], r) \preceq \Omega(\mathsf{succ}([q_2 \mapsto 0, q_4 \mapsto 1], r), r)$.

For all $S \subseteq \mathcal{F}$ and $\sigma \in \Sigma_2 \cup \Sigma_1$, we denote by $\mathsf{Pre}(S,\sigma) = \{F \mid \mathsf{succ}(F,\sigma) \in S\}$ the set of predecessors of S. The set of predecessors of a closed set $\downarrow F$ is closed and has a unique maximal element $\Omega(F,\sigma)$:

Lemma 4 *For all $F \in \mathcal{F} \cap \mathsf{safe}$ and $\sigma \in \Sigma_2 \cup \Sigma_1$, $\mathsf{Pre}(\downarrow F, \sigma) = \downarrow \Omega(F,\sigma)$.*

Proof Let $H \in \mathsf{Pre}(\downarrow F, \sigma)$. Hence $\mathsf{succ}(H,\sigma) \preceq F$. By Prop. 3$(i)$, we have $\Omega(\mathsf{succ}(H,\sigma),\sigma) \preceq \Omega(F,\sigma)$, from which we get $H \preceq \Omega(F,\sigma)$, by Prop. 3$(iii)$.
Conversely, let $H \preceq \Omega(F,\sigma)$. By Prop. 3$(ii)$, $\mathsf{succ}(H,\sigma) \preceq \mathsf{succ}(\Omega(F,\sigma),\sigma)$. Since by Prop. 3$(iv)$, $\mathsf{succ}(\Omega(F,\sigma),\sigma) \preceq F$, we get $\mathsf{succ}(H,\sigma) \preceq F$. □

We can now use the previous result to compute the controllable predecessors:

Proposition 4 *Let A be a* tbUKCW. *Given two antichains L_1, L_2 such that $L_1 \subseteq \mathcal{F}_2 \cap$ safe and $L_2 \subseteq \mathcal{F}_1 \cap$ safe:*

$$\mathsf{Pre}_1(\downarrow L_1) = \bigcup_{\sigma\in\Sigma_1} \mathsf{Pre}(\downarrow L_1, \sigma) = \bigcup_{\sigma\in\Sigma_1} \downarrow\{\Omega(F,\sigma) \mid F \in L_1\}$$
$$\mathsf{Pre}_2(\downarrow L_2) = \bigcap_{\sigma\in\Sigma_2} \mathsf{Pre}(\downarrow L_2, \sigma) = \bigcap_{\sigma\in\Sigma_2} \downarrow\{\Omega(F,\sigma) \mid F \in L_2\}$$

$\mathsf{Pre}_1(\downarrow L_1)$ can be computed in time $O(|\Sigma_1| \times |A| \times |L_1|)$, and $\mathsf{Pre}_2(\downarrow L_2)$ can be computed in time $O((|A| \times |L_2|)^{|\Sigma_2|})$.

As stated in the previous proposition, the complexity of our algorithm for computing the Pre_2 is worst-case exponential. We establish as a corollary of the next proposition that there is no polynomial times algorithm for computing Pre_2 unless $P{=}NP$. Given a graph $G = (V, E)$, a set of vertices W is independent iff no pairs of elements in W are linked by an edge in E. We denote by $\mathsf{IND}(G) = \{W \subseteq V \mid \forall\{v, v'\} \in E \cdot v \notin W \vee v' \notin W\}$ the set of independent sets in G. The problem "independent set" asks given a graph $G = (V, E)$ and an integer $0 \leq k \leq |V|$, if there exists an independent set in G of size larger than k. It is known to be NP-complete.

Proposition 5 *Given a graph $G = (V, E)$, we can construct in deterministic polynomial time a* UKCW *A, with $K = 0$, and an antichain L such that $\mathsf{IND}(G) = \downarrow\mathsf{Pre}_2(\mathsf{Pre}_1(\mathsf{Pre}_1((L))))$.*

Proof We start by a simple remark. Let A be tbUKCW with input states Q_2 and output states Q_1. When $k = 0$, Player 2's locations in $G(A, k)$ are exactly the subsets of Q_2 and Player 1's locations are the subsets of Q_1, and the partial order $\preceq$ corresponds to set inclusion.

Now, let us consider for each $e = \{v, v'\} \in E$ the antichain (for $\subseteq$) $L_{\{v,v'\}} = \{V \setminus \{v\}, V \setminus \{v'\}\}$, L compactly represents all the subsets of V that are independent of the edge $\{v, v'\}$. Clearly $\mathsf{IND}(G) = \bigcap_{\{v,v'\}\in E} \downarrow L_{\{v,v'\}}$. As a direct consequence of the NP completeness of the independent set problem, there cannot exist a polynomial time algorithm to compute the antichain for this intersection unless $P = NP$. Indeed, this antichain contains the maximal independent sets.

Now, we show how to construct a UKCW A and an antichain of subsets of states L such that $\mathsf{Pre}_2(\mathsf{Pre}_1(\mathsf{Pre}_1(L))$ is exactly $\mathsf{IND}(G)$. We can assume that V is totally ordered by some order, and for all edges $e = \{v, v'\} \in E$, we denote by $\pi_1(e)$ the minimal element of e and by $\pi_2(e)$ its maximal element. Now, the set of state of the automaton is structured in four layers: $S_3 = \{ok, ko\}$ belongs to Player 2, $S_2 = \{(v, e, i) \mid v \in V, e \in E, i \in \{1, 2\}\}$ belongs to Player 1, $S_1 = \{(v, e) \mid v \in V, e \in E\}$ belongs to Player 1. Finally, $S_0 = \{v \mid v \in V\}$ belongs to Player 2. Note that we do not make players strictly alternate here to simplify the exposition, it is easy to add a layer between the two actions of Player 1 to make the automaton turn-based. In those additional states, Player 2 would have only one action and the operation Pre_2 would simulate the identity. The objective of Player 1 is to ensure that the control ends up in state $ok \in S_3$, so we take $L = \{\{ok\}\}$. Now, we explain how to put transitions between states and compute $\mathsf{Pre}_2(\mathsf{Pre}_1(\mathsf{Pre}_1(((L)))$. The transitions from S_2 and S_3 are $\{((v, e, i), (e, i), ok) \mid \pi_i(e) \neq v\} \cup \{((v, e', i), (e, i), ko) \mid \pi_i(e) = v \vee e \neq e')\}$, it is easy to verify that $\mathsf{Pre}_1(L)$ is equal to $\downarrow\cup_{e\in E}\{\{(v, e, 1) \mid v \neq \pi_1(e)\}, \{(v, e, 2) \mid v \neq \pi_2(e)\}$. The transitions from S_1 to S_2 are $\{((v, e), i, (v, e, i)) \mid i \in \{1, 2\} \wedge v \in V \wedge e \in E\}$. As states of S_1 belongs to Player 1 the controllable configurations $\mathsf{Pre}_1(\mathsf{Pre}_1((L))$ are $\downarrow\cup_{e\in E}\{\{(v, e) \mid v \neq \pi_1(e)\}, \{(v, e) \mid v \neq \pi_2(e)\}\}$. The transitions from S_0 to S_1 are $\{(v, e, (v, e)) \mid v \in V \wedge e \in E\}$. As states in S_0 belongs to Player 2, we have that $\downarrow \mathsf{Pre}_2(\mathsf{Pre}_1(\mathsf{Pre}_1((L)))) = \mathsf{IND}(G)$, indeed Player 2 can decide to verify any edge for independence, so only independent set of vertices can be in $\mathsf{Pre}_2(\mathsf{Pre}_1(\mathsf{Pre}_1((L)))$. □

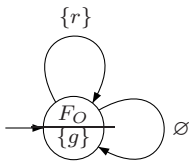


Fig. 2 Moore machine

Corollary 1 *There is no polynomial time algorithm to compute the* Pre_2 *operation on antichains unless* $P = NP$.

Note that this negative result is not a weakness of antichains. Indeed, it is easy to see from the proofs of those results that any algorithm based on a data structure that is able to represent compactly the set of subsets of a given set has this property.

Synthesis If a UCW A is realizable, it is easy to extract from the greatest fixpoint computation a Moore machine that realizes it. Let $\mathcal{F}_2^*$ and $\mathcal{F}_1^*$ be the two sets obtained by the greatest fixpoint computation. In particular, $\mathcal{F}_2^*$ and $\mathcal{F}_1^*$ are downward-closed and $\mathsf{Pre}_1(\mathcal{F}_2^*) = \mathcal{F}_1^*$, $\mathsf{Pre}_2(\mathcal{F}_1^*) = \mathcal{F}_2^*$. By definition of Pre_1, for all $F \in \lceil \mathcal{F}_1^* \rceil$, there exists $\sigma_F \in \Sigma$ such that $\mathsf{succ}(F, \sigma_F) \in \mathcal{F}_2^*$, and this σ_F can be computed. From this we can extract a Moore machine whose set of states is $\lceil \mathcal{F}_1^* \rceil$, the output function maps any state $F \in \lceil \mathcal{F}_1^* \rceil$ to σ_F, and the transition function, when reading some $\sigma \in \Sigma_2$, maps F to a state $F' \in \lceil \mathcal{F}_1^* \rceil$ such that $\mathsf{succ}(\mathsf{succ}(F, \sigma_F), \sigma) \preceq F'$ (it exists by definition of the fixpoint and by monotonicity of succ). The initial state is some state $F \in \lceil \mathcal{F}_1^* \rceil$ such that $F_0 \preceq F$ (it exists if the specification is realizable). Let M be this Moore machine. For any word w accepted by M, it is clear that w is also accepted by $\det(A, K)$, as succ is monotonic and $\mathcal{F}_1^* \subseteq \mathsf{safe}$. Therefore $L(M) \subseteq L_{\mathrm{uc},0}(\det(A, K)) = L_{\mathrm{uc},K}(A) \subseteq L_{\mathrm{uc}}(A)$.

Theorem 6 *Let* $\mathcal{F}_2^*$ *and* $\mathcal{F}_1^*$ *be the two sets obtained by the greatest fixpoint computation of the backward algorithm, if* $F_0 \in \mathcal{F}_1^*$*, then there exists a Moore machine with less than* $|\lceil \mathcal{F}_1^* \rceil|$ *states that encodes a winning strategy for Player 1 in the underlying safety game.*

Example We apply the antichain algorithm on the game of Figure 1(b). Remember that $I = \{r\}$ and $O = \{g\}$, so that $\Sigma_2 = \{\varnothing, \{r\}\}$ and $\Sigma_1 = \{\varnothing, \{g\}\}$. We denote counting functions in brackets and omit the states that map to -1. Note that some counting functions of the game do not appear in the picture as they are not reachable. We start the computation from the set of safe positions $S_1 = \{[q_1 \mapsto 0, q_3 \mapsto 1], [q_1 \mapsto 0, q_3 \mapsto 0], [q_1 \mapsto 0]\}$ represented by the antichain $\{[q_1 \mapsto 0, q_3 \mapsto 1]\}$. One application of Pre_2 on S_1 returns the set $S_2 = \downarrow [q_2 \mapsto 0, q_4 \mapsto 1]$. Then one application of Pre_1 on S_2 returns the set S_1. Therefore, we reach the fixpoint S_2 in one application of CPre.

From this fixpoint, we can compute a Moore machine. Let $F_1 = [q_1 \mapsto 0, q_3 \mapsto 1]$ and $F_2 = [q_2 \mapsto 0, q_4 \mapsto 1]$. Notice that $S_0 =\downarrow F_1$ and $S_1 =\downarrow F_2$. The Moore machine has only one state F_1. We then look at controller moves from F_1 that lead to a winning position. There is only one move $\{g\}$, which leads to the position $[q_2 \mapsto 0]$. This position is subsumed by F_2, from which whatever the environment does, the next position is F_1. Therefore there is a loop from F_1 to F_1 in the Moore machine, for the two possible moves of the environment: $\{r\}$ and $\varnothing$. It is depicted in Figure 2. Therefore the controller strategy obtained with our procedure is to always output a grant.

6.2 Forward algorithm for safety games with antichains

The forward algorithm of Section 3 can be used to solve the game $G(A, k)$. As for the backward algorithm, it is not necessary to construct the game explicitly. Indeed, during the execution of the OTFUR algorithm, the successors of a position F, which is a counting function, can be computed on demand via the successor function succ. Compared to the backward algorithm, the forward algorithm has the following advantage: it computes only the winning positions F (for Player 1) which are reachable from the initial position. We now show how to optimize this algorithm with antichains. We describe several optimizations.

Optimization 1: antichain of losing positions Let denote by $\mathcal{L}$ the set of losing positions for Player 1 in $G(A, k)$. Clearly, this set is upward closed. Indeed if Player 1 is not able to win in some position F, then she cannot win from any position where one or several counters of F have been increased. Therefore $\mathcal{L}$ can be represented by the antichain of its minimal elements. We can compute $\mathcal{L}$ incrementally during the execution of the OTFUR algorithm. For this we maintain an antichain L of (minimal) losing positions computed so far, i.e. at each step of the algorithm $L = \lfloor \{F \mid \mathsf{Losing}[F] = true\} \rfloor$. This set is updated each time a new position is known to be losing. We can use this information to prune the search space. Indeed, when we pick an edge (s, s') in the waiting list, if s' has never been visited before and $s' \in \uparrow L$, then we can directly set $\mathsf{Losing}[s'] = true$ and we do not need to add the successors of s' in the waiting list. If s' has never been visited and $s' \notin \uparrow L$, then among the successors s'' of s', we add in the waiting list only those which are not in $\uparrow L$. Finally if s' has already been visited, we check that $s \in \uparrow L$. In this case we do not need to inspect the successors of s.

Optimization 2: minimal and maximal successors Let F be a position of $G(A, k)$ owned by Player 1 (the controller). Clearly, F is losing (for the controller) iff all its minimal successors are losing. We get the dual of this property when F is a position owned by Player 2 (the environment). In this case F is losing (for the controller) iff one of its maximal successors is losing. Therefore to decide whether a position is losing, depending on whether it is a controller or an environment position, we have to visit its minimal or its maximal successors only. At line 11, this is done by adding to the waiting list only the edges (s', s'') such that s'' is a minimal (or maximal) successor of s'. In the case of a position owned by the controller, we can do even better. Indeed, we can add only one minimal successor in the waiting list at a time. If it turns out that this successor is losing, we add another minimal successor. Among the minimal successors, the choice is done as follows: we prefer to add an edge (s', s'') such that s'' has already been visited. Indeed, this potentially avoids unnecessary developments of new parts of the game.

We call FORWARD_ALL the method which consists of the OTFUR algorithm with optimization 1. We call FORWARD_EXI the method which consists of the OTFUR algorithm with optimizations 1 and 2. The difference between the two methods in the context of a compositional reasoning is discussed in depth in the following section. Informally, FORWARD_ALL computes the set of all reachable winning positions of the game, while FORWARD_EXI computes only a subset of it, as some pruning methods are used. However this subset is sufficient to decide whether the formula is realizable or not.

7 Performance Evaluation on the Monolithic Approach

In this section, we briefly present our implementation Acacia and compare it to Lily [15]. Acacia is a prototype implementation of the backward and forward antichain algorithms for LTL synthesis. To achieve a fair comparison, Acacia is written in Perl as Lily. Given an LTL formula and a partition of its propositions into inputs and outputs, Acacia tests realizability of the formula. If it is realizable, it outputs a Moore machine representing a winning strategy for the output player[10], otherwise it outputs a winning strategy for the input player. As Lily, Acacia runs in two steps. The first step builds a tbUCW for the formula, and the second step checks realizability of the automaton. As Lily, we borrow the LTL-to-tbUCW construction procedure from *Wring* [30] and adopt the automaton optimizations from Lily, so that we can exclude the influence of automata construction to the performance comparison between Acacia and Lily.[11]

Comparison with Kupferman-Vardi's Approach (implemented in Lily*)* In [19], the authors give a Safraless procedure for LTL synthesis. It is a three steps algorithm: (i) transform an LTL formula into a universal co-Büchi tree automaton (UCT) A that accepts the winning strategies of the system, (ii) transform A into an alternating weak tree automaton B (AWT) such that $L(B) \neq \varnothing$ iff $L(A) \neq \varnothing$, (iii) transform B into an equivalent Büchi tree automaton C (NBT) and test its emptiness. This latter problem can be seen as solving a game with a Büchi objective. This approach differs from our approach in the following points. First, in [19], the authors somehow reduce the realizability problem to a game with a *Büchi objective*, while our approach reduces it to a game with a *safety objective*. Second, our approach allows one to define a natural partial order on states that can be exploited by an antichain algorithm, which is not obvious in the approach of [19]. Finally, in [19], states of AWT are equipped with unique ranks that partition the set of states into layers. States which share the same rank are either all accepting or all non-accepting. The transition function allows one to stay in the same layer or to go in a layer with lower rank. A run is accepting if it gets stuck in a non-accepting layer. While our notion of counters looks similar to ranks, it is different. Indeed, the notion of rank does not constrain the runs to visit accepting states a bounded number of times (bounded by a constant). This is why a Büchi acceptance condition is needed, while counting the number of visited accepting states allows us to define a safety acceptance condition. Therefore the bound k we use in our approach and the rank are different notions. It is possible to construct examples for which our bound is exponentially larger than the rank of Lily.

[10] Note that the correctness of this Moore machine can be automatically verified by model-checking tools if desired.

[11] In Lily, this first step produces universal co-Büchi tree automata over Σ_1-labeled Σ_2-trees, which can easily be seen as tbUCWs over inputs Σ_2 and outputs Σ_1. Although the two models are close, we introduced tbUCWs for the sake of clarity (as all our developments are done on construction for word automata).

					Lily		k	Backward		Forward_ALL		Forward_EXI	
Examples	Realizable (y/n)	Formula size	tbUCW states	tbUCW Time(s)	rank	Check time(s)		|Moore machine|	Check time(s)	|Moore machine|	Check time(s)	|Moore machine|	Check time(s)
3	y	32	20	0.49	1	0.12	0	2	0.00	2	0.01	2	0.01
5	y	41	26	0.72	1	0.27	0	2	0.00	3	0.02	3	0.01
6	y	45	37	1.22	1	0.42	0	3	0.02	4	0.04	5	0.02
7	y	47	22	0.60	1	0.13	0	2	0.00	3	0.02	3	0.01
8	y	11	7	0.04	1	0.01	0	1	0.00	1	0.00	1	0.00
9	y	21	13	0.13	1	0.02	1	2	0.01	3	0.00	3	0.00
10	y	19	18	0.28	1	0.05	0	1	0.01	1	0.00	1	0.00
12	y	19	14	0.14	1	0.03	0	1	0.00	2	0.01	1	0.00
13	y	11	7	0.00	1	0.01	1	2	0.00	3	0.01	2	0.01
14	y	29	14	0.11	1	0.01	1	3	0.00	2	0.01	2	0.01
15	y	35	16	0.06	1	0.01	2	6	0.02	6	0.02	6	0.00
16	y	60	21	0.22	1	0.60	3	20	0.31	17	0.22	17	0.07
17	y	47	17	0.16	1	0.13	2	7	0.04	6	0.19	3	0.03
18	y	77	22	0.34	1	1.02	2	19	0.21	18	1.82	11	0.11
19	y	33	18	0.31	3	1.86	2	3	0.01	4	0.06	4	0.01
20	y	71	105	2.67	1	0.56	0	1	0.01	6	0.30	2	0.01
21	y	99	27	7.38	1	0.88	0	25	0.22	64	0.34	64	0.28
22	y	58	45	7.08	1	0.51	1	4	0.03	4	0.06	4	0.02
23	y	21	14	0.02	1	0.02	0	2	0.00	3	0.01	3	0.00
1	n	25	16	0.11 (0.08)	-	0.01	1	1	0.00	2	0.00	2	0.01
2	n	25	43	0.78 (0.07)	-	0.01	3	1	0.02	7	0.07	6	0.01
4	n	35	55	1.37 (0.86)	1	0.28	2	3	0.03	10	0.11	7	0.02
11	n	13	9	0.02 (0.64)	-	0.01	0	1	0.00	4	0.00	4	0.00
3.2	y	56	36	0.94	1	3.42	0	4	0.02	4	0.18	2	0.00
3.3	y	80	56	1.80	1	226.82	0	8	0.15	8	3.37	2	0.02
3.4	y	104	84	3.12	-	>t	0	16	1.24	16	70	2	0.04
3.5	y	128	128	4.74	-	>t	0	32	9.94	32	1808	2	0.14
3.6	y	152	204	10.2	-	>t	0	64	100	-	>m	2	0.46
3.7	y	176	344	26.5	-	>t	0	128	660	-	>m	2	2.35

Table 1 Performance comparison on tests included in Lily

Results We carried out experiments on a Linux platform with a 3.2GHz CPU (Intel i7) and 12GB of memory. We compared Acacia and Lily on the test suite included in Lily, and on other examples derived from Lily's examples, as detailed in the sequel. Let consider Table 1. *Formula size* gives the sizes of the formulas. They correspond to the number of atomic propositions and connectives. For realizability tests, Lily and Acacia construct the same tbUCW. However to check unrealizability, the methods differ, which results in different tbUCW. Indeed, while we check realizability by the environment of the negated specification, where the system does the first move, Lily checks realizability by the environment of the negated specification, where the environment does the first move. Therefore Lily takes as input the negated formula where every occurrence of an output signal σ is replaced by $\mathcal{X}\sigma$. The number of states (column tbUCW *state*) as well as the time to construct the tbUCW

(column tbUCW *Time(s)*) are given in the table (for unrealizable specifications, we put this time in parenthesis for Lily). We consider different algorithms for testing realizability: Lily, the backward antichain algorithm, and two versions of the forward algorithm, FORWARD_ALL and FORWARD_EXI respectively, described in the previous section. For all the methods, we give the time to check for realizability (column *Check time(s)*). The column |Moore machine| gives the size of the synthesized Moore machine. We also give the *rank* used in Lily and the bound k (column k) needed for our methods (it is common to the three methods we use). Finally, the timeout is fixed to 1 hour, and is denoted by $> t$. When the execution runs out of memory, we denote it by $> m$.

Comments Lily's test suite includes examples 1 to 23. Except examples 1, 2, 4, and 11, they are all realizable. In the test suite, demo 3 describes a scheduler. We have taken a scalability test by introducing more clients. In Table 1, from 3.4 to 3.7, when the number of clients reached 4, Lily ran over-time (> 3600 seconds). However, Acacia managed in finishing the check within the time bound. Clearly, the FORWARD_EXI method is most efficient. When considering Lily's examples 1 to 23 and the total time to check realizability (including the tbUCW construction), the improvement in time complexity compared to Lily is less impressive, because the time to construction the automaton is often much bigger than the realizability check time. However it was not expected that this first step would have been the bottleneck of the approach. Indeed in the classical approach of [24], the foreseen bottleneck is clearly the second step, which relies on Safra's determinization. In the following sections, we will focus on a compositional approach that overcomes this problem, as indeed in this approach the formulas are divided into smaller specifications for which we can construct the automaton independently.

8 Compositional Safety Games and LTL Synthesis

In this section, we define compositional safety games and develop two compositional algorithms to solve such games. Compositional reasoning on safety games are supported by the existence of a most permissive strategy, that we formalized under the notion of *master plan*.

Master plan Let $G = (S_1, S_2, \Gamma_1, \Delta_1, \Delta_2)$ be a safety game on a finite set of moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$. Let $W^\natural$ be the winning positions of G for Player 1. Let $\Lambda_1 : S_1 \to 2^{\mathsf{Moves}_1}$ be defined as follows: for all $s \in S_1$, $\Lambda_1(s) = \{m \in \Gamma_1(s) \mid \Delta_1(s,m) \in W^\natural\}$ i.e., $\Lambda_1(s)$ contains all the moves that Player 1 can play in s in order to win the safety game. We call Λ_1 the *master plan* of Player 1 and we write it $\mathsf{MP}(G)$. The following lemma states that $\mathsf{MP}(G)$ can be interpreted as a compact representation of all the winning strategies of Player 1 in the game G:

Lemma 5 *For all strategies λ_1 of Player 1 in G, for all $s \in S$, λ_1 is winning in G from s iff λ_1 is a strategy in $(G[\mathsf{MP}(G)], s)$ and $\lambda_1(s) \neq \bot$.*

The master plan associated with a game can be computed in a backward fashion by using variants of the CPre operator and sequence W defined in Section 3. The variant of CPre considers the effect of some Player 1's move followed by some Player 2's move. Let $\widehat{\mathsf{CPre}} : (S_1 \to 2^{\mathsf{Moves}_1}) \to (S_1 \to 2^{\mathsf{Moves}_1})$ be defined as follows. For all $s \in S_1$, let:

$$\widehat{\mathsf{CPre}}(\Lambda)(s) = \{m \in \Lambda(s) \mid \forall m' \in \mathsf{Moves}_2 : \Lambda(\Delta_2(\Delta_1(s,m), m')) \neq \varnothing\}$$

Consider the following sequence of functions: $\Lambda_0 = \Gamma_1$, and $\Lambda_i = \widehat{\mathsf{CPre}}(\Lambda_{i-1})$, $i \geq 1$. This sequence stabilizes after at most $O(|S|)$ iterations and we denote by $\Lambda^\natural$ the function on which the sequence stabilizes. Clearly, the value on which the sequence stabilizes corresponds exactly to the master plan of G:

Theorem 7 $\Lambda^\natural = \mathsf{MP}(G)$.

Composition of safety games We now consider products of safety games. Let G^i, $i \in \{1, \ldots, n\}$, be n safety games $G^i = (S_1^i, S_2^i, \Gamma_1^i, \Delta_1^i, \Delta_2^i)$ defined on the same sets of moves $\mathsf{Moves} = \mathsf{Moves}_1 \uplus \mathsf{Moves}_2$. Their product, denoted by $\otimes_{i=1}^{i=n} G^i$, is the safety game $G^\otimes = (S_1^\otimes, S_2^\otimes, \Gamma_1^\otimes, \Delta_1^\otimes, \Delta_2^\otimes)$[12] defined as follows:

- $S_j^\otimes = S_j^1 \times S_j^2 \times \cdots \times S_j^n$, $j = 1, 2$;
- for $s = (s^1, s^2, \ldots, s^n) \in S_1^\otimes$, $\Gamma_1^\otimes(s) = \Gamma_1^1(s^1) \cap \Gamma_1^2(s^2) \cap \cdots \cap \Gamma_1^n(s^n)$;
- for $j \in \{1, 2\}$ and $s = (s^1, s^2, \ldots, s^n) \in S_j^\otimes$, let $m \in \Gamma_1^\otimes(s)$ if $j = 1$ or $m \in \mathsf{Moves}_2$ if $j = 2$. Then $\Delta_j^\otimes(s) = (t^1, t^2, \ldots, t^n)$, where $t^i = \Delta_j^i(s^i, m)$ for all $i \in \{1, 2, \ldots, n\}$;

Backward compositional reasoning We now define a backward compositional algorithm to solve the safety game $G^\otimes$. The correctness of this algorithm is justified by the following simple lemmas. For readability, we express the properties for composed games defined from two components. All the properties generalize to any number of components. The first part of the lemma states that to compute the master plan of a composition, we can first reduce each component to its *local* master plan. The second part of the lemma states that the master plan of a component is the master plan of the component where the choices of Player 1 has been restricted by one application of the $\widehat{\mathsf{CPre}}$ operator.

Lemma 6 (a) *Let* $G^{12} = G^1 \otimes G^2$, *let* $\Lambda_1 = \mathsf{MP}(G^1)$ *and* $\Lambda_2 = \mathsf{MP}(G^2)$ *then*

$$\mathsf{MP}(G^{12}) = \mathsf{MP}(G^1[\Lambda_1] \otimes G^2[\Lambda_2])$$

(b) *For any game* G, $\mathsf{MP}(G) = \mathsf{MP}(G[\widehat{\mathsf{CPre}}(\Gamma_1)])$.

Let $\Lambda : S_1^1 \times S_1^2 \times \cdots \times S_1^n \to 2^{\mathsf{Moves}}$, we let $\pi_i(\Lambda)$ the function with domain S_1^i and co-domain 2^{Moves_1} such that for all $s \in S_1^i$, $\pi_i(\Lambda)(s)$ is the set of moves allowed by Λ in one tuple $(s^1, s^2, \ldots, s^n)$ such that $s^i = s$. Formally, $\pi_i(\Lambda)(s) = \bigcup\{\Lambda(s^1, s^2, \ldots, s^n) \mid (s^1, s^2, \ldots, s^n) \in S_1^\otimes,\ s^i = s\}$. Given two functions $\Lambda_1 : S_1 \to 2^{\mathsf{Moves}_1}$ and $\Lambda_2 : S_1 \to 2^{\mathsf{Moves}_1}$, we define $\Lambda_1 \cap \Lambda_2$ as the function on domain S_1 such that for all $s \in S_1$: $\Lambda_1 \cap \Lambda_2(s) = \Lambda_1(s) \cap \Lambda_2(s)$. Given two functions $\Lambda_1 : S_1 \to 2^{\mathsf{Moves}_1}$ and $\Lambda_2 : S_2 \to 2^{\mathsf{Moves}_1}$, we define $(\Lambda_1 \times \Lambda_2) : S_1 \times S_2 \to 2^{\mathsf{Moves}_1}$ as $(\Lambda_1 \times \Lambda_2)(s_1, s_2) = \Lambda_1(s_1) \cap \Lambda_2(s_2)$.

Based on Lemma 6, we propose the following compositional algorithm (Algo 2) to compute the master plan of a safety game defined as the composition of local safety games. First, compute locally the master plans of the components. Then compose the local master plans and apply one time the $\widehat{\mathsf{CPre}}$ operator to this composition. This application of $\widehat{\mathsf{CPre}}$ compute a new function Λ that contains information about the one-step inconsistencies between local master plans. Project back on the local components the information gained by the function Λ, and iterate. Correctness is asserted by Theorem 8.

[12] Clearly, the product operation is associative up to isomorphism.

Algorithm 2: Backward composition

Data: $G^{\otimes} = G^1 \otimes G^2 \otimes \cdots \otimes G^n$
$\Lambda \leftarrow \Gamma_1^{\otimes}$;
repeat
  $\Lambda^i := \mathsf{MP}(G^i[\pi_i(\Lambda)]), 1 \leq i \leq n$;
  $\Lambda := \widehat{\mathsf{CPre}}(\Lambda \cap (\Lambda^1 \times \cdots \times \Lambda^n))$
until Λ *does not change*;
return Λ

Theorem 8 *The value Λ returned by Algorithm 2 is equal to* $\mathsf{MP}(G^{\otimes})$.

Forward compositional reasoning When solving safety games, we may be interested only in computing winning strategies for a fixed starting position, say s_{ini}. In this case, the value of the master plan is not useful for positions that are not reachable when playing winning strategies from s_{ini}. So, we are interested in computing a master plan only for the winning and *reachable* positions. Given a game G and a state s_{ini}, we denote by $\mathsf{Reach}(G, s_{\mathsf{ini}})$ the subset of positions that are reachable from s_{ini} in G i.e., the position s' such that there exists a finite sequence $s_0 s_1 \dots s_n$ with $s_0 = s_{\mathsf{ini}}$, $s_n = s'$ and for all i, $0 \leq i < n$, there exists $m \in \Gamma_1(s_i) \cup \mathsf{Moves}_2$ such that $s_{i+1} = \Delta(s_i, m)$. The *master plan of reachable positions* for (G, s_{ini}), denoted by $\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})$ is defined for all $s \in S$ as follows:

$$\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})(s) = \begin{cases} \mathsf{MP}(G)(s) & \text{if } s \in \mathsf{Reach}(G[\Lambda], s_{\mathsf{ini}}) \\ \varnothing & \text{otherwise.} \end{cases}$$

The following lemma states that for a game defined compositionally, its master plan can also be defined compositionally. For readability we express the lemma only for two components but, as for the previous lemmas, it extends to any number of components:

Lemma 7 *Let $\Lambda_1 = \mathsf{MP}_{\mathsf{Reach}}(G^1, s^1_{\mathsf{ini}})$ and $\Lambda_2 = \mathsf{MP}_{\mathsf{Reach}}(G^2, s^2_{\mathsf{ini}})$.*

$$\mathsf{MP}_{\mathsf{Reach}}(G^1 \otimes G^2, (s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}})) = \mathsf{MP}_{\mathsf{Reach}}(G^1[\Lambda_1] \otimes G^2[\Lambda_2], (s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}}))$$

Based on the previous lemma, Algorithm 3 shows how to compute the master plan of reachable positions of a safety game defined compositionally.

Algorithm 3: Forward composition

Data: $G^{\otimes} = G^1 \otimes G^2 \otimes \cdots \otimes G^n$
$\Lambda^i := \mathsf{MP}_{\mathsf{Reach}}(G^i, s^i_{\mathsf{ini}}), 1 \leq i \leq n$;
$\Lambda := \mathsf{MP}_{\mathsf{Reach}}(G^1[\Lambda^1] \otimes \cdots \otimes G^n[\Lambda^n], (s^1_{\mathsf{ini}}, s^2_{\mathsf{ini}}, \dots, s^n_{\mathsf{ini}}))$
return Λ

As composition of safety games is an associative operator, we can use variants of Algorithm 3 above where we first compose some of the components and compute their master plan of reachable positions before doing the global composition.

To efficiently compute the master plan of reachable positions of a game G, we can use the OTFUR algorithm defined in Section 6 (Algo 1). Indeed, at the end of the algorithm, the master plan which allows all moves that lead to a winning position is exactly $\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})$. Fig. 3 illustrates the result of the OTFUR algorithms applied on the product of two safety games G_1, G_2 over the possible moves o_1, o_2, o_3 for Player 1 and i_1, i_2 for

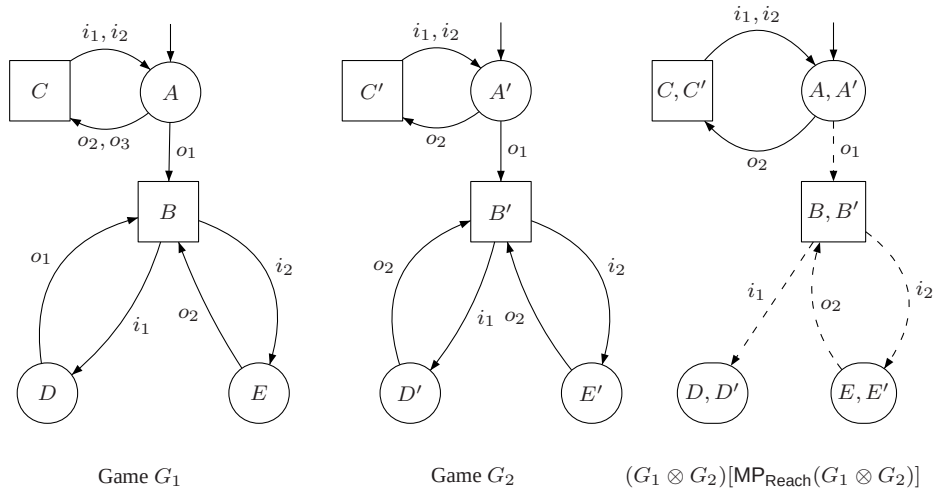


Fig. 3 Two games and their common master plan of reachable positions

Player 2. We assume that G_1, G_2 contains only winning actions, i.e. $G_i = G_i[\mathsf{MP}(G_i)]$ for $i = 1, 2$. The master plan of reachable positions for $G_1 \otimes G_2$ corresponds to plain arrows. Dashed arrows are those which have been traversed during the OTFUR algorithm but have been removed due to back-propagation of information about losing positions. From node $\langle A, A' \rangle$ the move o_3 is not a common move, therefore o_3 is not available in the product. However o_2 is available in both games and leads to C and C' respectively. Similarly, o_1 is available in both games and goes to $\langle B, B' \rangle$. From $\langle B, B' \rangle$ one can reach $\langle D, D' \rangle$ by i_1 but from $\langle D, D' \rangle$ there is no common action. Therefore $\langle D, D' \rangle$ is unsafe. Since one of the successor of $\langle B, B' \rangle$ is unsafe and $\langle B, B' \rangle$ is owned by Player 2, $\langle B, B' \rangle$ is declared to be unsafe as well. All the remaining moves are winning in the $G_1 \otimes G_2$, as they are winning both in G_1 and G_2.

Remark 1 It should be noted that each A^i in Alg. 2 can be replaced by the full *master plan* without changing the output of the forward algorithm. Indeed, it is easy to see that $\mathsf{Reach}(G[\mathsf{MP}_{\mathsf{Reach}}(G, s_{\mathsf{ini}})], s_{\mathsf{ini}}) = \mathsf{Reach}(G[\mathsf{MP}(G)], s_{\mathsf{ini}})$. So, we can mix the backward and forward algorithms. For instance, we can compute locally the master plan of each G^i using the backward algorithm of section 6, and then check global realizability using the OTFUR algorithm.

Compositional LTL Synthesis We show how to define compositionally the safety game associated with an LTL formula when this formula is given as a conjunction of subformulas i.e., $\psi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$. We first construct for each subformula ϕ_i the corresponding tbUKCW A_{ϕ_i} on the alphabet of ψ[13], and their associated safety games $G(\phi_i, K)$. The game $G(\psi, K)$ for the conjunction ψ is isomorphic to the game $\otimes_{i=1}^{i=n} G(\phi_i, K)$.

To establish this result, we rely on a notion of product at the level of turn-based automata. Let $A_i = (\Sigma_2, \Sigma_1, Q_2^i, Q_1^i, q_0^i, \alpha^i, \delta_2^i, \delta_1^i)$ for $i \in \{1, 2\}$ be two turn-based automata, then their product $A_1 \otimes A_2$ is the turn-based automaton defined as $(\Sigma_2, \Sigma_1, Q_2^1 \uplus Q_2^2, Q_1^1 \uplus$

[13] It is necessary to keep the entire alphabet when considering the subformulas to ensure proper definition of the product of games that asks for components defined on the same set of moves.

$Q_1^2, Q_{\mathsf{ini}}^1 \uplus Q_{\mathsf{ini}}^2, \alpha_1 \uplus \alpha_2, \delta_2^1 \uplus \delta_2^2, \delta_1^1 \uplus \delta_1^2$). As we use universal interpretation i.e., we require all runs to respect the accepting condition, it is clear that executing the $A_1 \otimes A_2$ on a word w is equivalent to execute both A_1 and A_2 on this word. So w is accepted by the product iff it is accepted by each of the automata.

Proposition 6 *Let A_1 and A_2 be two* UCW *on the alphabet $\Sigma_1 \uplus \Sigma_2$, and $K \in \mathbb{N}$: (i) $L_{uc}(A_1 \otimes A_2) = L_{uc}(A_1) \cap L_{uc}(A_2)$, (ii) $L_{uc,K}(A_1 \otimes A_2) = L_{uc,K}(A_1) \cap L_{uc,K}(A_2)$*

As the state space and transition relation of $A_1 \otimes A_2$ is the disjunct union of the space spaces and transition relations of A_1 and A_2, the determinization of $A_1 \otimes A_2$ for a fixed $K \in \mathbb{N}$ is equivalent to the synchronized product of the determinizations of A_1 and A_2 for that K, and so we get the following theorem.

Theorem 9 *Let $\psi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, $K \in \mathbb{N}$, $G(\psi, K)$ is isomorphic to $\otimes_{i=1}^{i=n} G(\phi_i, K)$.*

Assume from now on that we have fixed some $K \in \mathbb{N}$. In order to solve the game $G(\psi, K)$, in practice we first solve locally the subgames $G(\phi_i, K)$ by computing their master plans, or their reachable master plans, and then compute the master plan of their composition. In particular, to compute the master plan of a local game, we can use the backward algorithm described in Section 6 with the optimizations based on antichains. Indeed, the antichain of winning positions provides a compact representation of the master plan. In particular, let F be some Player 1's position in some safety game $G(\psi, K)$ and let W be the antichain of maximal winning positions (for Player 1). Then:

$$\mathsf{MP}(G(\psi, K))(F) \;=\; \begin{cases} \varnothing & \text{if } F \notin {\downarrow}W \\ \{\sigma \in \Sigma_1 \mid \mathsf{succ}(F, \sigma) \in {\downarrow}W\} & \text{otherwise} \end{cases}$$

To compute the master plan of reachable positions, we can use the forward OTFUR algorithm. However only the optimization which maintains an antichain of losing positions (see Section 6) can be used in the forward algorithm. Indeed, the optimization based on the minimal and maximal successors are used to prune the search space. Although it works when one wants to decide if there is a winning strategy, it is not correct to use it for computing the master plan (which is a representation of all the winning strategies) as some parts of the safety game may be ignored by the pruning strategy. In practice, we can use the FORWARD_ALL or BACKWARD algorithms for the local intermediate games and the FORWARD_EXI algorithm for the global game.

Dropping assumptions Even if it is natural to write large LTL specifications as conjunctions of subformulas $\phi_1 \wedge \cdots \wedge \phi_n$, formulas of the following form are often used:

$$(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow (\bigwedge_{j=1}^{j=m} \phi_j)$$

where ψ_i's formalize a set of assumptions made on the environment (Player 2) and ϕ_j's formalize a set of guarantees that the system (Player 1) must enforce. In this case, we rewrite the formula into the logical equivalent formula

$$\bigwedge_{j=1}^{j=m} ((\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j)$$

which is a conjunction of LTL formulas as needed for the compositional construction described above. As logical equivalence is maintained, realizability is maintained as well. The formulas of the form $\bigwedge_{j=1}^{j=m}((\bigwedge_{i=1}^{i=n}\psi_i) \rightarrow \phi_j)$ are larger than the original formula as assumptions are duplicated. However the subformulas $(\bigwedge_{i=1}^{i=n}\psi_i) \rightarrow \phi_j, j \in \{1,\ldots,m\}$ are usually such that to guarantee ϕ_j, Player 1 does not need all the assumptions on the left of the implication. It is thus tempting to remove those assumptions that are *locally* unnecessary in order to get smaller local formulas. In practice, we apply the following rule. Let $\psi_1 \wedge \psi_2 \rightarrow \phi$ be a local formula such that ψ_2 and ϕ do not share common propositions then we replace $\psi_1 \wedge \psi_2 \rightarrow \phi$ by $\psi_1 \rightarrow \phi$. This simplification is correct in the following sense: if the formula obtained after dropping some assumptions in local formulas is realizable then the original formula is also realizable. Further, a Player 1's strategy to win the game defined by the simplified formula is also a Player 1's strategy to win the game defined by the original formula. This is justified by the fact that the new formula logically implies the original formula i.e. $\psi_1 \rightarrow \phi$ logically implies $\psi_1 \wedge \psi_2 \rightarrow \phi$. However, this heuristic is not complete because the local master plans may be more restrictive than necessary as we locally forget about global assumptions that exist in the original formula. We illustrate this on two examples.

Let $I = \{\mathsf{req}\}$, $O = \{\mathsf{grant}\}$ and $\phi = (\Box\Diamond\mathsf{req}) \rightarrow \Box\Diamond\mathsf{grant}$. In this formula, the assumption $\Box\Diamond\mathsf{req}$ is not relevant to the guarantee $\Box\Diamond\mathsf{grant}$. Realizing ϕ is thus equivalent to realizing $\Box\Diamond\mathsf{grant}$. However, the set of strategies realizing ϕ is not preserved when dropping the assumption. Indeed, the strategy that outputs a grant after each req realizes ϕ but it does not realize $\Box\Diamond\mathsf{grant}$, as this strategy relies on the behavior of the environment. Thus dropping assumption is weaker than the notion of *open implication* of [13], which requires that the strategies realizing ϕ have to realize $\Box\Diamond\mathsf{grant}$.

As illustrated by the previous example, dropping assumption does not preserve the set of strategies that realize the formula. Therefore, it can be the case that a realizable formula cannot be shown realizable with our compositional algorithm after locally dropping assumptions. In addition, it can be the case that a formula becomes unrealizable after dropping local assumptions. Consider for instance the formula $\phi = \Box\Diamond\mathsf{req} \rightarrow (\Box\Diamond\mathsf{grant} \wedge \Box(\mathcal{X}(\neg\mathsf{grant})\ \mathcal{U}\ \mathsf{req}))$. This formula is realizable, for instance by the strategy which outputs a grant iff the environment signal at the previous tick was a req. Other strategies realize this formula, like those which grant a request every n req signal (n is fixed), but all the strategies that realize ϕ have to exploit the behavior of the environment. Thus there is no strategy realizing the conjunction of $\Box\Diamond\mathsf{grant}$ and ϕ. Consequently, when we decompose ϕ into $\Box\Diamond\mathsf{req} \rightarrow \Box\Diamond\mathsf{grant}$ and $\Box\Diamond\mathsf{req} \rightarrow \Box(\mathcal{X}(\neg\mathsf{grant})\ \mathcal{U}\ \mathsf{req})$, we must keep $\Box\Diamond\mathsf{req}$ in the two formulas.

Nevertheless, in our experiments, the dropping assumption heuristic is very effective and almost always maintains *compositional realizability*.

9 Experiments: Compositional Approach

As the monolithic methods, the compositional algorithms have been implemented in our prototype ACACIA. The performances are evaluated on the examples provided with the tool LILY and on a larger specification of a buffer controller inspired by the IBM rulebase tutorial [14]. Some results are reported on Table 3.

Lily's test cases and a parametric example First, we compare the performances of the new compositional algorithms with the best results of the monolithic approach, which are the

ones obtained via the monolithic FORWARD_EXI method. This is done on Lily's realizable examples among 1 to 23 and the parametric examples 3.1 to 3.7. For the compositional approach, we use the backward fixpoint algorithm to solve the local safety games, and the forward method FORWARD_EXI to solve the global game. The formula tested for the comparison with previous works are of moderate size. It should be noted that for larger formulas, it is often not possible to construct the UCW at all, and so the monolithic algorithm are not applicable[14].

In those benchmarks, the formulas are of the form $\bigwedge_{i=1}^{i=n} \psi_i \rightarrow \bigwedge_{j=1}^{j=m} \phi_j$ where $\bigwedge_{i=1}^{i=n} \psi_i$ are a set of *assumptions* and $\bigwedge_{j=1}^{j=m} \phi_j$ are a set of *guarantees*. We decompose such formula into several pieces $(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j$, as described in the previous section. We consider different methods for checking realizability: a monolithic approach and two compositional approaches. The second realizability method is the same as the first but includes the dropping assumption (DA) heuristic when it is applicable. For each of the method, we give the size of the constructed automata and the time to construct them. We also give the time for realizability checking (including the strategy synthesis), and finally the total time (the best total time is in bold face). Let us mention that when applying our compositional algorithm, we construct a tbUCW for each conjunct $(\bigwedge_{i=1}^{i=n} \psi_i) \rightarrow \phi_j$. Therefore the column $|\mathsf{tbUCW}|$ refers to the size of tbUCW by monolithic approach and $\Sigma_i(|\mathsf{tbUCW}_i|)$ refers to the sum of sizes of tbUCW corresponding to the sub-specifications of the decomposition.

On small examples, we can see that the benefit of the compositional approach is not big (and in some cases the monolithic approach is even better). However for bigger formulas (demo 3.2 to 3.7), decomposing the formulas decreases the time to construct the automata, and the total realizability time is therefore better.

Now, we evaluate the benefit of dropping assumptions (last group of columns). For those experiments, we only consider the subset of formulas for which this heuristic can be applied. Our dropping heuristic does not work for demo 9 as it becomes unrealizable after the application of dropping assumptions. As we see in the table, the benefit of dropping assumptions is important and is growing with the size of the formulas that are considered. The compositional algorithms outperform the monolithic ones when combined with dropping assumptions. They also show promises for better scalability. This is confirmed by our next benchmark.

A realistic case study Now, we consider a set of realistic formulas. All those formulas are out of reach of the monolithic approach as even the Büchi automaton for the formula cannot be constructed with state of the art tools. The generalized buffer (GenBuf) originates from the IBM's tutorial for her RuleBase verification tool. The benchmark has also the nice property that it can be scaled up by increasing the number of receivers in the protocol. A detailed description of the case study is given in appendix as well as our LTL formalization. In this case study, the formulas are of the form $\bigwedge_{i=1}^{i=n} \psi_i \rightarrow \phi_i$ and so they are readily amenable to our compositional algorithms.

In this case study, formulas are large: for example, the sum of the number of states in the UCW of the components is 96 for $\mathsf{gb}(s_2, r_2)$, and 2399 states for $\mathsf{gb}(s_2, r_7)$. Note that the tool Wring cannot handle $\mathsf{gb}(s_2, r_2)$ monolithically.

This case study allows us to illustrate the effect of different strategies for exploiting associativity of the product operation. In particular, we use different ways of parenthesizing the local games. In all those examples, the local games and intermediate combination of local games are solved with the backward antichain algorithm, while the last compositional

[14] This is the case for the larger formulas in the IBM case study below.

	Monolithic FORWARD_EXI FORWARD_EXI				Compositional FORWARD_EXI(global) BACKWARD(local)				Compositional + DA FORWARD_EXI(global) BACKWARD(local)			
examples	|tbUCW| (states)	tbUCW Time(s)	Check Time(s)	Total time(s)	Σ_i |tbUCW$_i$|	tbUCW Time(s)	Check Time(s)	Total time(s)	Σ_i |tbUCW$_i$|	tbUCW Time(s)	Check Time(s)	Total time(s)
3	20	0.49	0.01	0.5	28	0.40	0.01	0.41	17	0.06	0.00	**0.06**
5	26	0.71	0.01	0.72	42	0.70	0.02	0.72	34	0.40	0.02	**0.42**
6	37	1.22	0.02	1.24	57	1.14	0.03	1.17	45	0.79	0.06	**0.85**
7	22	0.60	0.01	0.61	41	0.66	0.02	0.68	33	0.40	0.02	**0.42**
9	13	0.13	0.00	**0.13**	31	0.26	0.00	0.26	na	na	na	na
13	7	0.00	0.01	0.01	4	0.01	0.00	**0.01**	na	na	na	na
14	14	0.11	0.01	**0.12**	27	0.77	0.01	0.78	15	0.03	0.00	0.03
15	16	0.06	0.00	**0.06**	22	0.11	0.03	0.14	na	na	na	na
16	21	0.22	0.07	**0.29**	45	0.20	0.14	0.34	na	na	na	na
17	17	0.16	0.03	**0.19**	23	0.16	0.05	0.21	na	na	na	na
18	22	0.34	0.11	**0.45**	45	0.35	0.16	0.51	na	na	na	na
19	18	0.31	0.01	0.32	27	0.25	0.03	0.28	27	0.26	0.01	**0.27**
20	105	2.67	0.01	2.68	154	2.43	0.03	2.46	101	1.52	0.02	**1.54**
21	27	7.38	0.28	7.66	43	1.40	0.52	1.92	44	0.55	0.51	**1.06**
22	45	7.08	0.02	7.1	80	10.26	0.05	10.31	49	1.51	0.13	**1.64**
3.2	36	0.94	0.00	0.94	40	0.79	0.02	**0.81**	na	na	na	na
3.3	56	1.80	0.02	1.82	60	1.21	0.06	**1.27**	na	na	na	na
3.4	84	3.12	0.04	3.16	80	1.63	0.10	**1.73**	na	na	na	na
3.5	128	3.52	0.12	3.64	100	2.04	0.17	**2.21**	na	na	na	na
3.6	204	10.22	0.46	10.68	120	2.40	0.39	**2.79**	na	na	na	na
3.7	344	26.48	2.35	28.82	140	2.96	1.02	**3.98**	na	na	na	na

Table 2 Performance comparison of the different compositional algorithms implemented in Acacia on Lily's benchmark

step (at the top) is done with the FORWARD_EXI method. In each strategy we first compute the master plan of each sub-formula. Then the column Flat refers to the strategy that check global realizability directly. The column Binary refers to the strategy that computes global realizability incrementally using the binary tree of sub-formulas. Finally, the column Heuristic refers to the strategy that computes global realizability incrementally using a specific tree of sub-formula defined by the user[15]. The column UCW_OPT refers to the time to optimize the automata with Lily's optimizations (this time was included in the UCW time in Table 2).

The last column in Table 2 gives the size of the Moore machines that are constructed by our algorithm. Those machines are small when compared to the tbUCW from which they are constructed. For example, the Moore machine that encodes a winning strategy for Player 1 in the largest example (gb_s2_r7) has 149 states while the tbUCW for the specification has 2399 states. This is striking as in theory, the winning strategies could be exponentially larger than the tbUCW of their specification.

[15] The input language of our tool allow us to specify sub-formula (spec-units) and composition rules to guide the incremental construction of the global winning strategy, see Appendix.

					FLAT	BINARY	HEURISTIC	
	k	Σ|tbUCW|	tbUCW Time(s)	UCW_OPT Time(s)	Check Time(s)	Check Time(s)	Check Time(s)	|Moore machine|
gb_s2_r2	2	91	4.83	0.08	0.84	0.99	0.98	54
gb_s2_r3	2	150	8.52	0.17	7.33	36.27	6.99	63
gb_s2_r4	2	265	15.64	0.53	36.88	125.60	24.19	86
gb_s2_r5	2	531	26.48	2.11	154.02	266.36	70.41	107
gb_s2_r6	2	1116	50.70	14.38	889.12	1164.44	335.44	132
gb_s2_r7	2	2399	92.01	148.46	2310.74	>t	1650.83	149

Table 3 Performance comparison on a scalability test for the forward methods

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Proceedings of 16th International Colloquium on Automata, Languages, and Programming (ICALP), *Lecture Notes in Computer Science*, vol. 372, pp. 1–17. Springer-Verlag (1989)
2. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Proceedings of the 9th International Conference on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 1466, pp. 163–178. Springer-Verlag (1998)
3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from psl. Electronic Notes in Theoretical Computer Science **190**(4), 3–16 (2007)
4. Boker, U., Kupferman, O.: Co-ing büchi made tight and useful. In: Proceedings of the 24th IEEE Annual Symposium on Logic in Computer Science (LICS), pp. 245–254. IEEE Computer Society (2009)
5. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Proceedings of the 16th International Conference on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 3653, pp. 66–80. Springer-Verlag (2005)
6. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 4144, pp. 17–30. Springer-Verlag (2006)
7. Doyen, L., Raskin, J.F.: Improved algorithms for the automata-based approach to model-checking. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 4424, pp. 451–465. Springer-Verlag (2007)
8. Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 6015, pp. 2–22. Springer-Verlag (2010)
9. Ehlers, R.: Symbolic bounded synthesis. In: Proceedings of the 22nd International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 6174, pp. 365–379. Springer-Verlag (2010)
10. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 5643, pp. 263–277. Springer-Verlag (2009)
11. Filiot, E., Jin, N., Raskin, J.F.: Compositional algorithms for LTL synthesis. In: Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA), *Lecture Notes in Computer Science*, vol. 6252, pp. 122–127. Springer-Verlag (2010)
12. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games: A Guide to Current Research, *Lecture Notes in Computer Science*, vol. 2500. Springer-Verlag (2002)
13. Greimel, K., Bloem, R., Jobstmann, B., Vardi, M.Y.: Open implication. In: Proceedings of the 35th International Colloqium on Automata, Languages and Programming (ICALP), *Lecture Notes in Computer Science*, vol. 5126, pp. 361–372. Springer-Verlag (2008)
14. IBM: Rulebase tutorial, available at `www.haifa.ibm.com/projects/verification/rb_homepage/tutorial3/`

15. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Proceedings of the 6th International Conference on Formal Methods in Computer Aided Design (FMCAD), pp. 117–124. IEEE Computer Society (2006)
16. Kuijper, W., van de Pol, J.: Compositional control synthesis for partially observable systems. In: Proceedings of the 20th International Conference on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 5710, pp. 431–447. Springer-Verlag (2009)
17. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 4144, pp. 31–44. Springer-Verlag (2006)
18. Kupferman, O., Vardi, M.Y.: On bounded specifications. In: Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *Lecture Notes in Computer Science*, vol. 2250, pp. 24–38. Springer-Verlag (2001)
19. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), pp. 531–542. IEEE Computer Society (2005)
20. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points. In: Proceedings of the 25th International Colloqium on Automata, Languages and Programming (ICALP), *Lecture Notes in Computer Science*, vol. 1443, pp. 53–66. Springer-Verlag (1998)
21. Martin, D.: Borel determinacy. Annals of Mathematics **102**, 363–371 (1975)
22. Piterman, N.: From nondeterministic büchi and streett automata to deterministic parity automata. Logical Methods in Computer Science **3**(3) (2007)
23. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 3855, pp. 364–380. Springer-Verlag (2006)
24. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), pp. 179–190. ACM, ACM Press (1989)
25. Raskin, J.F., Chatterjee, K., Doyen, L., Henzinger, T.A.: Algorithms for omega-regular games with imperfect information. Logical Methods in Computer Science **3**(3) (2007)
26. Rosner, R.: Modular synthesis of reactive systems. Ph.d. dissertation, Weizmann Institute of Science (1992)
27. S. Safra: On the complexity of ω automata. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), pp. 319–327. IEEE Computer Society (1988)
28. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA), *Lecture Notes in Computer Science*, vol. 4762, pp. 474–488. Springer-Verlag (2007)
29. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for ltl games. In: Proceedings of the 9th International Conference on Formal Methods in Computer Aided Design (FMCAD), pp. 77–84. IEEE Computer Society (2009)
30. Somenzi, F., Bloem, R.: Efficient büchi automata from LTL formulae. In: Proceedings of the 12th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 1855, pp. 248–263. Springer-Verlag (2000)
31. Thomas, W.: Church's problem and a tour through automata theory. In: Pillars of Computer Science, *Lecture Notes in Computer Science*, vol. 4800, pp. 635–655. Springer-Verlag (2008)

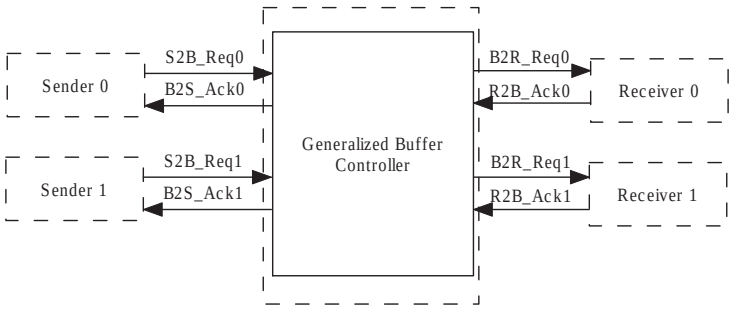


Fig. 4 Diagram of a buffer controller connected with 2 senders and 2 receivers

A Description of the Generalized Buffer Controller

The generalized buffer (GenBuf) originates from the IBM's tutorial for her RuleBase verification tool.

Figure. 4 illustrates the interface of a controller. To focus on its control flow, we abstract away the data buses *DI* and *DO*. And we do not connect it with a FIFO which is present in Anzu. Except those, our controller shares with RuleBase and Anzu the same interface.

The interface between GenBuf and the senders is a 4-phase handshaking protocol described below:

1. When a sender, say sender i, has data to send, it initiates a transfer by asserting $s2b\_req(i)$ (Server to Buffer Request). One cycle later, the sender puts the data on its data bus.
2. When GenBuf can service the sender, it reads the data from the data bus and asserts $b2s\_ack(i)$.
3. In the cycle following the assertion of $b2s\_ack(i)$, the sender should deassert the signal $s2b\_req(i)$. From this point onwards, the data on the data bus is considered invalid.
4. The end of the transaction is marked by GenBuf deasserting $b2s\_ack(i)$. A new transaction may begin one cycle after the deassertion of $b2s\_ack(i)$. GenBuf may hold $b2s\_ack(i)$ asserted for several cycles before eventually deasserting it.

The protocol between GenBuf and the receivers is similar, except that the GenBuf initiates the data transfer and it guarantees round-robin scheduling on the requests to the receivers.

Table. 4 lists the behavior properties of GenBuf and its environment.

Properties of the behavior between GenBuf and the senders.

Assumption 1	The initial value of $s2b\_req(i)$ is low. $s2b\_req(i) = 0$
Assumption 2	A request is not lowered until it is served. $\Box((s2b\_req(i) = 1 \wedge b2s\_ack(i) = 0) \rightarrow s2b\_req(i) = 1)$
Assumption 3	In the cycle following the assertion of $b2s\_ack(i)$, the sender should deassert the signal $s2b\_req(i)$. $\Box(b2s\_ack(i) = 1 \rightarrow \mathcal{X}(s2b\_req(i) = 0))$
Guarantee 1	The initial value of $b2s\_ack(i)$ is low. $b2s\_ack(i) = 0$
Guarantee 2	When GenBuf can service the sender, it asserts $b2s\_ack(i)$. Immediate acknowledge is forbidden. Because the data of the sender are not valid until one step after the assertion of the request. A Request from a sender shall always be acknowledged. $\Box(\ (s2b\_req(i) = 0 \wedge \mathcal{X}(s2b\_req(i) = 1))$ $\rightarrow \mathcal{X}(b2s\_ack(i) = 0 \wedge \Diamond(b2s\_ack(i) = 1)))$
Guarantee 3	There is no acknowledge without a request. $\Box((b2s\_ack(i) = 0 \wedge \mathcal{X}(s2b_reeq(i) = 0)) \rightarrow \mathcal{X}(b2s\_ack(i) = 0))$
Guarantee 4	Only one sender sends data at any one time. $\Box(\bigvee_i \bigwedge_{j \neq i} b2s\_ack(j) = 0)$

Properties of the behavior between GenBuf and the receivers.

Assumption 4	The initial value of $r2b\_ack(i)$ is low. $r2b\_ack(i) = 0$
Assumption 5	There is no acknowledge without a request. $\Box(b2r\_req(i) = 0 \rightarrow \mathcal{X}(r2b\_ack(i) = 0))$
Assumption 6	A request from the buffer is always acknowledged. $\Box(b2r\_req(i) = 1 \rightarrow \mathcal{X}(\Diamond(r2b\_ack(i) = 1)))$
Guarantee 5	The initial value of $b2r\_req(i)$ is low. $b2r\_req(i) = 0$
Guarantee 6	A request is not lowered until it is served. $\Box((b2r\_req(i) = 1 \wedge r2b\_ack(i) = 0) \rightarrow \mathcal{X}(b2r\_req(i) = 1))$
Guarantee 7	GenBuf will deassert its request to receiver i one cycle after the receiver i acknowledged the request. $\Box(r2b\_ack(i) = 1 \rightarrow \mathcal{X}(R2B\_Req(i) = 0))$
Guarantee 8	GenBuf will not make two consecutive requests to any receiver, and guarantee round-robin scheduling. Suppose there are two receivers. $\Box(\ (R2B\_Req(0) = 1 \wedge \mathcal{X}(R2B\_Req(0) = 0))$ $\rightarrow \mathcal{X}(R2B\_Req(0) = 0\ \mathcal{U}\ (R2B\_Req(0) = 0 \wedge R2B\_Req(1) = 1)))$ $\Box(\ (R2B\_Req(1) = 1 \wedge \mathcal{X}(R2B\_Req(1) = 0))$ $\rightarrow \mathcal{X}(R2B\_Req(1) = 0\ \mathcal{U}\ (R2B\_Req(1) = 0 \wedge R2B\_Req(0) = 1)))$
Guarantee 9	GenBuf does not request two receivers simultaneously. $\Box(\bigvee_i \bigwedge_{j \neq i} b2r\_req(j) = 0)$

Property linking the sender side and the receiver side.

Guarantee 10	A request from the senders will always trigger a request to the receivers. $\Box((\bigvee_i s2b\_req(i) = 1) \rightarrow \mathcal{X}(\Diamond(\bigvee_j b2r\_req(j) = 1)))$

Table 4 Behavior properties of GenBuf and its environment

We present the formulas of $\mathtt{gb}(s_2, r_2)$ as an example for the file format of Acacia'10. In this example, we define 4 components in terms of `spec_unit`. If all of them are realizable, the clause directed by `group_order` instructs Acacia'10 to continue along the parenthesized groups, like `(sb_0 br_1)` and `(sb_1 br_0)`. The process halts whenever an unrealizable subgroup is encounted. The final step is to check the group which includes all the components.

```
################################################
[spec_unit sb_0]
```

```
################################################
assume s2b_req0=0;
assume G((s2b_req0=1 * b2s_ack0=0) -> X(s2b_req0=1));
assume G(b2s_ack0=1 -> X(s2b_req0=0));

b2s_ack0=0;
G((s2b_req0=0 * X(s2b_req0=1)) -> X(b2s_ack0=0 * X(F(b2s_ack0=1))));
G((b2s_ack0=0 * X(s2b_req0=0)) -> X(b2s_ack0=0));
G(b2s_ack0=0 + b2s_ack1=0);

################################################
[spec_unit sb_1]
################################################
assume s2b_req1=0;
assume G((s2b_req1=1 * b2s_ack1=0) -> X(s2b_req1=1));
assume G(b2s_ack1=1 -> X(s2b_req1=0));

b2s_ack1=0;
G((s2b_req1=0 * X(s2b_req1=1)) -> X(b2s_ack1=0 * X(F(b2s_ack1=1))));
G((b2s_ack1=0 * X(s2b_req1=0)) -> X(b2s_ack1=0));
G(b2s_ack0=0 + b2s_ack1=0);

################################################
[spec_unit br_0]
################################################
assume r2b_ack0=0;
assume G(b2r_req0=0 -> X(r2b_ack0=0));
assume G(b2r_req0=1 -> X(F(r2b_ack0=1)));

b2r_req0=0;
G(r2b_ack0=1 -> X(b2r_req0=0));
G((b2r_req0=1 * r2b_ack0=0) -> X(b2r_req0=1));
G((b2r_req0=1 * X(b2r_req0=0)) -> X(b2r_req0=0 U (b2r_req0=0 * b2r_req1=1)));
G((b2r_req0=0) + (b2r_req1=0) );
G((s2b_req0=1 + s2b_req1=1) -> X(F(b2r_req0=1 + b2r_req1=1)));

################################################
[spec_unit br_1]
################################################
assume r2b_ack1=0;
assume G(b2r_req1=0 -> X(r2b_ack1=0));
assume G(b2r_req1=1 -> X(F(r2b_ack1=1)));

b2r_req1=0;
G(r2b_ack1=1 -> X(b2r_req1=0));
G((b2r_req1=1 * r2b_ack1=0) -> X(b2r_req1=1));
G((b2r_req1=1 * X(b2r_req1=0)) -> X(b2r_req1=0 U (b2r_req1=0 * b2r_req0=1)));
G((b2r_req0=0) + (b2r_req1=0) );
G((s2b_req0=1 + s2b_req1=1) -> X(F(b2r_req0=1 + b2r_req1=1)));

group_order = (sb_0 br_1) (sb_1 br_0);
```