Synthesising Succinct Strategies in Safety Games*

Gilles Geeraerts, Joël Goossens, and Amélie Stainer

Université libre de Bruxelles, Département d'Informatique, Brussels, Belgium

Abstract. Finite turn-based safety games have been used for very different problems such as the synthesis of linear temporal logic (LTL) [8], the synthesis of schedulers for computer systems running on multiprocessor platforms [4], and also for the determinisation of timed automata [3]. In these contexts, games are implicitly defined, and their size is at least exponential in the size of the input. Nevertheless, there are natural relations between states of arenas of such games. We first formalise the properties that we expect on the relation between states, thanks to the notion of alternating simulation. Then, we show how such simulations can be exploited to (1) improve the running time of the OTFUR algorithm[6] to compute winning strategies and (2) obtain a succinct representation of a winning strategy. We also show that our general theory applies to the three applications mentioned above.

1 Introduction

Finite, turn-based, safety games are arguably one of the most simple, yet relevant, classes of games. They are played by two players (A and B) on a finite graph (called the arena), whose set of vertices is partitioned into Player A and Player B vertices, (that we call A and B-states respectively). A play is an infinite path in this graph, and is obtained by letting the players move a token on the vertices. Initially, the token is on a designated initial vertex. At each round of the game, the player who owns the vertex marked by the token decides on which successor node to move it next. A play is winning for A if the token never touches some designated bad nodes, otherwise, it is winning for B.

Such games are a natural model to describe the interaction of a potential controller with a given environment, where the aim of the controller is to avoid the bad states that model system failures. In this framework, computing a winning strategy for the player amounts to *synthesising* a control policy that guarantees no bad state will be reached, no matter how the environment behaves. Safety

* This research has been supported by the Belgian F.R.S./FNRS FORESt grant, number 14621993.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n601148 (CASSTING)

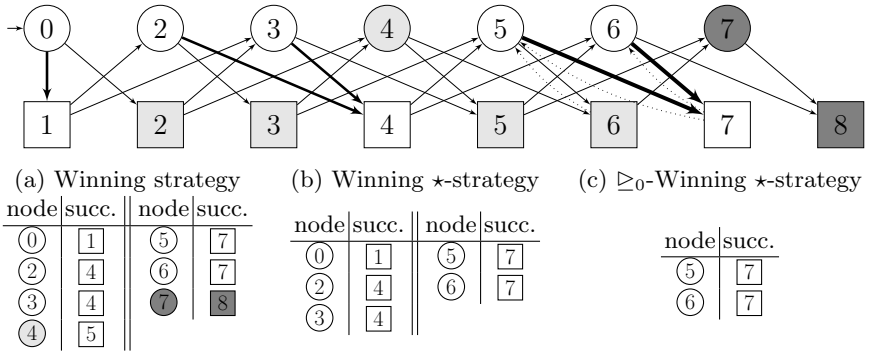

Fig. 1. Urn-filling Nim game with $N = 8$, and three winning strategies.

games have also been used as a tool to solve other problems such as LTL realisability [8], real-time scheduler synthesis [4] or timed automata determinisation [3]. We will come back to those applications.

To illustrate our ideas, we consider, throughout the paper, a variation of the well-known Nim game [5], where players have to *fill* an urn instead of removing balls from it. The game is played by two players (A and B) as follows. Initially, an heap of N balls is shared by the players, and the urn is empty. Then, the players play by turn and pick either 1 or 2 balls from the heap and put them into the urn. A player looses the game if he is the last to play (i.e., the heap is empty after he has played). An arena modeling this game (for $N = 8$) is given in Fig. 1 (top), where A-states are circles, B-states are squares, and the numbers labelling the states represent the number of balls *inside the urn*. The arena obtained from Fig. 1 *without the dotted edges* faithfully models the description of the urn-filling game we have sketched above (assuming Player A plays first). We say that a state is winning if, from this state, Player A has a *winning strategy*, i.e. he can always win the game whatever Player B does (and vice-versa for losing states). For instance, $\boxed{8}$ is loosing for Player A. Indeed, in this state, all 8 balls are now into the urn, hence the heap is empty. Moreover, $\boxed{8}$ belongs to Player B, hence Player A has necessarily played just before reaching it. State $\textcircled{7}$ can also be declared as losing, since Player A has no other choice than tossing the last ball into the urn. Thus, in this game, the objective of Player A is to avoid the set $\mathsf{Bad} = \{\textcircled{7}, \boxed{8}\}$. It is well-known [5] that a simple characterisation of the set of winning states[1] can be given. For each state v, let $\lambda(v)$ denote its label. Then, the winning states (in white in Fig 1) are all the A-states v s.t. $\lambda(v) \mod 3 \neq 1$ plus all the B-states v' s.t. $\lambda(v') \mod 3 = 1$.

One of the nice properties of safety games is that they admit *memory-less winning strategies*, i.e., if Player A has a winning strategy in the game, then he has a winning strategy that depends only on the current state (in other

[1] In order to make our example more interesting (this will become clear in the sequel), we have added the three *dotted edges* from $\boxed{7}$ to $\textcircled{6}$ and $\textcircled{5}$ respectively, and from $\boxed{6}$ to $\textcircled{5}$ although those actions are not permitted in the original game. However, observe that those extra edges do not modify the set of winning states.

words, this strategy is a function from the set of A-states to the set of B-states). Memory-less strategies are often regarded as simple and straightforward to implement (remember that the winning strategy is very often the actual control policy that we want to implement in, say, an embedded controller). Yet, this belief falls short in many practical applications such as the three mentioned above because the arena is not given explicitly, and its size is *at least exponential* in the size of the original problem instance. Hence, two difficulties arise, even with memory-less strategies. First, the computation of winning strategies might request the traversal of the whole arena, which might be intractable in practice. Second, a naive implementation of a winning strategy σ by means of an exponentially large table, mapping each A-state v to its safe successor $\sigma(v)$, is not realistic. For example, a winning strategy for our urn-filling game is given in Fig. 1 (a), and it contains one line for each A-state, even for the losing ones.

In this work, we consider the problem of computing *efficiently* winning strategies (for Player A) that can be *succinctly* represented. To formalise this problem, we consider $\star$-*strategies*, which are *partial functions* defined on a subset of A-states only. A $\star$-strategy can be regarded as an *abstract representation* of a family of (plain) strategies, that we call *concretisations* of the $\star$-strategies (they are all the strategies that agree with the $\star$-strategy). Then, we want to compute *succinct* $\star$-strategies that are defined on the smallest possible number of states, but that are still safe because all their concretisations are winning. In the example, a winning $\star$-strategy of minimal size is given in Fig. 1 (b). Remark that we can now omit two A-states (the losing ones), because those states are not reachable anyway, but we still need to remember what to play on all winning states. Unfortunately, we show (see Section 3) that, unless P=NP, no polynomial-time algorithm exists to compute minimal $\star$-strategies, because the associated decision problem is NP-complete. This holds when the arena is given explicitly, so the difficulty is exacerbated in practice, where the arena is much larger than the problem instance.

To cope with this complexity, we consider heuristics inspired from the *antichain* line of research [7]. Antichain techniques rely on a *simulation partial order* on the states of the system. This simulation is exploited to *prune* the state space that the algorithms need to explore, and to obtain *efficient data structures* to store the set of states that the algorithms need to maintain. These techniques have been applied to several relevant problems, such as LTL model-checking [7] or multi-processor schedulability [9] with remarkable performance improvements of several orders of magnitude.

While a general theory of antichain has been developed [7] in the setting of *automata-based models* (that are well-suited for verification), they have been scarcely applied in the setting of *games*. One notable exception is the aforementioned work of Filiot *et al.* on LTL realisability [8] where the problem of realisability is reduced to a safety game, and a simulation on the game states is exploited in an efficient algorithm tailored for those games. The heuristics are thus limited to this peculiar case, and their correctness cannot always be deduced from the notion of *simulation* but requires *ad hoc* proofs.

In this paper, we advocate the use of a different notion, i.e. *turn-based alternating simulations* (tba-simulations for short), which are adapted from the alternating simulations introduced in [2]. Tba-simulations allow us to develop an elegant and general theory, that extends the ideas of Filiot *et al.* and that is applicable to a broad class of safety games (including the three examples mentioned above). In our running example, a tba-simulation $\trianglerighteq_0$ exists and is given by: $v \trianglerighteq_0 v'$ iff v and v' belong to the same player, $\lambda(v) \geq \lambda(v')$ and $\lambda(v) \mod 3 = \lambda(v') \mod 3$. Then, it is easy to see that the winning strategy of Fig. 1 (a) exhibits some kind of *monotonicity* wrt $\trianglerighteq_0$: for instance ⑤ $\trianglerighteq_0$ ②, and the winning strategy consists in putting two balls in the urn in both cases. Hence, we can further reduce the representation of the strategy to the one in Fig. 1 (c). Indeed, while not all concretisations of this strategy are winning (for instance, the strategy does not say what to play in ③ and obviously going to 4 should be avoided), all concretisations compatible with $\trianglerighteq$ *are* winning. It is then easy to implement *succinctly* this strategy: only the table in Fig. 1 (c) needs to be stored, because he relation $\trianglerighteq$ can be directly computed on the description of the states. All these intuitions are formalised in Section 4, where we show that, in general, it is sufficient to store the strategy on the maximal *antichain* of the reachable winning states, i.e., on a set of states that are all incomparable by the tba-simulation, and thus very compact.

Next, we present, in Section 5 *an efficient on-the-fly algorithm to compute such succinct $\star$-strategies*, which is adapted from the classical OTFUR algorithm to solve safety games [6]. Our algorithm incorporates several heuristics that stem directly from the definition of tba-simulation (in particular, it incorporates an optimisation that was not present in the work of Filiot et al. [8]). Finally, in Section 6, we devise a criterion that allows to *determine when a simulation relation on a game arena is also a tba-simulation*. We show that the safety games one considers in three applications introduced above (LTL realisability, real-time feasibility and determinisation of timed automata) respect this criterion, which demonstrates the wide applicability of our approach.

2 Preliminaries

Turn-based safety games A *finite turn-based game arena* is a tuple $\mathcal{A} = (V_A, V_B, E, I)$, where V_A and V_B are the finite sets of states controlled by Players A and B respectively; $E \subseteq (V_A \times V_B) \cup (V_B \times V_A)$ is the set of edges; and $I \in V_A$ is the initial state. We denote by V the set $V_A \cup V_B$. Given a finite arena $\mathcal{A} = (V_A, V_B, E, I)$ and a state $v \in V$, we let $\mathsf{Succ}(\mathcal{A}, v) = \{v' \mid (v, v') \in E\}$ and $\mathsf{Reach}(\mathcal{A}, v) = \{v' \mid (v, v') \in E^*\}$, where E^* denotes the reflexive and transitive closure of E. We lift the definitions of Reach and Succ to sets of states in the usual way. A *finite turn-based safety game* is a tuple $G = (V_A, V_B, E, I, \mathsf{Bad})$ where (V_A, V_B, E, I) is a finite turn-based game arena, and $\mathsf{Bad} \subseteq V$ is the set of bad states that A wants to avoid. The definitions of Reach and Succ carry on to games: $\mathsf{Reach}((\mathcal{A}, \mathsf{Bad}), v) = \mathsf{Reach}(\mathcal{A}, v)$ and $\mathsf{Succ}((\mathcal{A}, \mathsf{Bad}), v) = \mathsf{Succ}(\mathcal{A}, v)$. When the game is clear from the context, we often omit it.

Plays and strategies During the game, players interact to produce a play, which is a finite or infinite path in the graph (V, E). Players play turn by turn, by moving a *token* on the game's states. Initially, the token is on state I. At each turn, the player who controls the state marked by the token gets to choose the next state. A *strategy* for A is a function $\sigma : V_A \to V_B$ such that for all $v \in V_A$, $(v, \sigma(v)) \in E$. We extend strategies to set of states S in the usual way: $\sigma(S) = \{\sigma(v) \mid v \in S\}$. A strategy σ for A is *winning for a state* $v \in V$ iff no bad states are reachable from v in the graph G_σ obtained from G by removing all the moves of A which are not chosen by σ, i.e. $\mathsf{Reach}\,(G_\sigma, v) \cap \mathsf{Bad} = \emptyset$, where $G_\sigma = (V_A, V_B, E_\sigma, I, \mathsf{Bad})$ and $E_\sigma = \{(v, v') \mid (v, v') \in E \wedge v \in V_A \implies v' = \sigma(v)\}$. We say that a strategy σ is *winning* in a game $G = (V_A, V_B, E, I, \mathsf{Bad})$ iff it is winning in G for I.

Winning states and attractors A state $v \in V$ in G is *winning* iff there exists a strategy σ that is winning in G for v. We denote by Win the set of winning states (for Player A). By definition, any strategy such that $\sigma(\mathsf{Win}) \subseteq \mathsf{Win}$ is thus winning. Moreover, it is well-known that the set Win of winning states can be computed in polynomial time (in the size of the arena), by computing the so-called *attractor* of the unsafe states. In a game $G = (V_A, V_B, E, I, \mathsf{Bad})$, the sequence $(\mathsf{Attr}_i)_{i \geq 0}$ of attractors (of the Bad states) is defined as follows. $\mathsf{Attr}_0 = \mathsf{Bad}$ and for all $i \in \mathbb{N}$, $\mathsf{Attr}_{i+1} = \mathsf{Attr}_i \cup \{v \in V_B \mid \mathsf{Succ}\,(v) \cap \mathsf{Attr}_i \neq \emptyset\} \cup \{v \in V_A \mid \mathsf{Succ}\,(v) \subseteq \mathsf{Attr}_i\}$. It is well-known that, for all $i \geq 0$, $v \in \mathsf{Attr}_i$ means that, from v, Player B can force the game to visit Bad in at most i steps. For finite games, the sequence stabilises after a finite number of steps on a set of states that we denote $\mathsf{Attr}_{\mathsf{Bad}}$. Then, v belongs to $\mathsf{Attr}_{\mathsf{Bad}}$ iff Player B can force the game to visit Bad from v. Thus, the set of winning states for player A is $\mathsf{Win} = V \setminus \mathsf{Attr}_{\mathsf{Bad}}$. It is then straightforward to compute a winning strategy σ: for all $v \in V_A \cap \mathsf{Win}$, we let $\sigma(v) = v'$, where $v' \in \mathsf{Win}$. Such a node is guaranteed to exist by definition of the attractor.

Partial orders, closed sets and antichains Fix a finite set S. A relation $\trianglerighteq \in S \times S$ is a partial order iff $\trianglerighteq$ is reflexive, transitive and antisymmetric, i.e. for all $s \in S$: $(s, s) \in \trianglerighteq$ (reflexivity); for all $s, s', s'' \in S$, $(s, s') \in \trianglerighteq$ and $(s', s'') \in \trianglerighteq$ implies $(s, s'') \in \trianglerighteq$ (transitivity); and for all $s, s' \in S$: $(s, s') \in \trianglerighteq$ and $(s', s) \in \trianglerighteq$ implies $s = s'$ (antisymmetry). As usual, we often write $s \trianglerighteq s'$ and $s \not\trianglerighteq s'$ instead of $(s, s') \in \trianglerighteq$ and $(s, s') \notin \trianglerighteq$, respectively. The $\trianglerighteq$-*downward closure* $\downarrow^{\trianglerighteq}(S')$ of a set $S' \subseteq S$ is defined as $\downarrow^{\trianglerighteq}(S') = \{s \mid \exists s' \in S', s' \trianglerighteq s\}$. Symmetrically, the *upward closure* $\uparrow^{\trianglerighteq}(S')$ of S' is defined as: $\uparrow^{\trianglerighteq}(S') = \{s \mid \exists s' \in S' : s \trianglerighteq s'\}$. Then, a set S' is *downward closed* (resp. *upward closed*) iff $S' = \downarrow^{\trianglerighteq}(S')$ (resp. $S' = \uparrow^{\trianglerighteq}(S')$). When the partial order is clear from the context, we often write $\downarrow(S)$ and $\uparrow(S)$ instead of $\downarrow^{\trianglerighteq}(S)$ and $\uparrow^{\trianglerighteq}(S)$ respectively. Finally, a subset α of some set $S' \subseteq S$ is an *antichain* on S' with respect to $\trianglerighteq$ if for all $s, s' \in \alpha$, $s \not\trianglerighteq s'$. An antichain α on S' is said to be a set of *maximal elements of* S' (or, simply *a maximal antichain* of S') iff for all $s_1 \in S'$ there is $s_2 \in \alpha$: $s_2 \trianglerighteq s_1$. Symmetrically, an antichain α on S' is a set of *minimal elements of* S' (or *a minimal antichain* of S') iff for all $s_1 \in S'$ there is $s_2 \in \alpha$: $s_1 \trianglerighteq s_2$. It is easy to check that if α and β are maximal and minimal antichains of S' respectively, then $\downarrow(\alpha) = \downarrow(S')$ and

$\uparrow(\beta) = \uparrow(S')$. Intuitively, α (β) can be regarded as a symbolic representation of $\downarrow(S')$ ($\uparrow(S')$), which is of minimal size in the sense that it contains no pair of $\trianglerighteq$-comparable elements. Moreover, since $\trianglerighteq$ is a partial order, each subset S' of the finite set S admits a unique minimal and a unique maximal antichain, that we denote by $\lfloor S' \rfloor$ and $\lceil S' \rceil$ respectively. Observe that one can always effectively build a $\lceil S' \rceil$ and $\lfloor S' \rfloor$, simply by iteratively removing from S', all the elements that are strictly $\trianglerighteq$-dominated by (for $\lceil S' \rceil$) or that strictly dominate (for $\lfloor S' \rfloor$) another one.

Simulation relations Fix an arena $G = (V_A, V_B, E, I, \mathsf{Bad})$. A relation $\trianglerighteq \subseteq V_A \times V_A \cup V_B \times V_B$ is a *simulation relation compatible*[2] *with* Bad (or simply a *simulation*) iff it is a partial order[3] and for all $(v_1, v_2) \in \trianglerighteq$: either $v_1 \in \mathsf{Bad}$ or: (i) for all $v_2' \in \mathsf{Succ}(v_2)$, there is $v_1' \in \mathsf{Succ}(v_1)$ s.t. $v_1' \trianglerighteq v_2'$ and (ii) $v_2 \in \mathsf{Bad}$ implies that $v_1 \in \mathsf{Bad}$. On our example, the relation $\trianglerighteq_0 = \{(v, v') \in V_A \times V_A \cup V_B \times V_B \mid \lambda(v) \geq \lambda(v') \text{ and } \lambda(v) \mod 3 = \lambda(v') \mod 3\}$ is a simulation relation compatible with $\mathsf{Bad} = \{\textcircled{7}, \boxed{8}\}$. Moreover, $\mathsf{Win} = \{v \in V_A \mid \lambda(v) \mod 3 \neq 1\} \cup \{v \in V_B \mid \lambda(v) \mod 3 = 1\}$ is downward closed for $\trianglerighteq_0$ and its complement (the set of losing states), is upward closed. Finally, Win admits a single maximal antichain for $\trianglerighteq_0$: $\mathsf{MaxWin} = \{\boxed{7}, \textcircled{6}, \textcircled{5}\}$.

3 Succinct strategies

Let us first formalise the notion of *succinct strategy*. As explained in the introduction, a naive way to implement a memory-less strategy σ is to store, in an appropriate data structure, the set of pairs $\{(v, \sigma(v)) \mid v \in V_A\}$, and implement a controller that traverses the whole table to find action to perform each time the system state is updated. While the definition of strategy asks that $\sigma(v)$ be defined for all A-states v, this information is sometimes indifferent, for instance, when v is not reachable in G_σ. Thus, we want to reduce the number of states v s.t. $\sigma(v)$ is crucial to keep the system safe.

$\star$-strategies We introduce the notion of $\star$-strategy to formalise this idea: a $\star$-strategy is a function $\hat{\sigma} : V_A \mapsto V_B \cup \{\star\}$, where $\star$ stands for a 'don't care' information. We denote by $\mathsf{Supp}(\hat{\sigma})$ the *support* $\hat{\sigma}^{-1}(V_B)$ of $\hat{\sigma}$, i.e. the set of nodes v s.t. $\hat{\sigma}(v) \neq \star$. Such $\star$-strategies can be regarded as a representation of a family of concrete strategies. A *concretisation* σ of a $\star$-strategy $\hat{\sigma}$ is a strategy σ s.t. for all $v \in V_A$, $\hat{\sigma}(v) \neq \star$ implies $\hat{\sigma}(v) = \sigma(v)$. A $\star$-strategy $\hat{\sigma}$ is *winning* if every concretisation of $\hat{\sigma}$ is winning (intuitively, $\hat{\sigma}$ is winning if A always wins when he plays according to $\hat{\sigma}$, whatever choices he makes when $\hat{\sigma}$ returns $\star$). The *size* of a $\star$-strategy $\hat{\sigma}(v)$ is the size of $\mathsf{Supp}(\hat{\sigma})$. On our running example, the strategy σ displayed in Fig. 1 (b) – assuming the lines where $\sigma(v) = \star$ have been omitted – is a winning $\star$-strategy of minimal size.

[2] See [8] for an earlier definition of a simulation relation compatible with a set of states.

[3] Observe that our results can be extended to the case where the relations are *preorders*, i.e. transitive and reflexive relations.

Computing succinct $\star$-strategies Our goal is to compute *succinct* $\star$-strategies, defined as $\star$-strategies of minimal size. To characterise the hardness of this task, we consider the following decision problem, and prove that it is NP-complete:

*Problem 1 (*MinSizeStrat*).* Given a finite turn-based safety game G and an integer $k \in \mathbb{N}$, decide whether there is a winning $\star$-strategy of size smaller than k in G.

Theorem 1. MinSizeStrat *is NP-complete.*

Proof. Let $X = \{x_1, \cdots, x_m\}$ be a set of m Boolean variables. Let $\phi = \wedge_{1 \leq i \leq n} C_i$ be a Sat formula (with n clauses) such that for all $1 \leq i \leq n$, $C_i = \vee_{1 \leq j \leq n_i} l^i_j$ with $l^i_j \in L = \{x_1, \cdots, x_m, \neg x_1, \cdots, \neg x_m\}$ (L is called the set of literals). Then, let us build a finite turn-based safety game G and an integer k in polynomial time as follows. We let $G = (V_A, V_B, E, I, \mathsf{Bad})$ where:

- $V_A = \{\text{init}^A\} \cup \mathcal{L}^A \cup X \cup C$, where $\mathcal{L}^A = \{x^A_1, \cdots, x^A_m, \neg x^A_1, \cdots, \neg x^A_m\}$, $\mathcal{X} = \{X_1, \cdots, X_m\}$ and $\mathcal{C} = \{C_1, \cdots, C_n\}$
- $V_B = \{\text{init}^B, \text{bad}\} \cup \{x^B_1, \cdots, x^B_m, \neg x^B_1, \cdots, \neg x^B_m\}$
- $E = \{(\text{init}^A, \text{init}^B)\} \cup \{\text{init}^B\} \times (\mathcal{X} \cup \mathcal{C})$
  $\cup \left\{ \begin{array}{c} (X_i, x^B_i), (X_i, \neg x^B_i), (x^B_i, x^A_i), (\neg x^B_i, \neg x^A_i), \\ (x^A_i, \text{init}^B), (\neg x^A_i, \text{init}^B), (x^A_i, \text{bad}), \\ (\neg x^A_i, \text{bad}), (X_i, \text{bad}), (C_i, \text{bad}) \end{array} \middle| 1 \leq i \leq m \right\}$
  $\cup (\mathcal{L}^A \cup \mathcal{C} \cup \mathcal{X}) \times \{\text{bad}\}$
  $\cup \{(C_i, x^B_h) \mid 1 \leq i \leq n, \exists 1 \leq j \leq n_i : l^i_j = x_h\}$
  $\cup \{(C_i, \neg x^B_h) \mid 1 \leq i \leq n, \exists 1 \leq j \leq n_i : l^i_j = \neg x_h\}$
- $I = \text{init}^A$
- $\mathsf{Bad} = \{\text{bad}\}$

Finally, let $k = 2 \times m + n$. An example of the construction for the formula $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ is given in Figure 2 (note that bad and the $(\neg)x^B_i$'s) have been duplicated to enhance readability). States of A are circles and states of B are squares.

Let us show that ϕ is satisfiable iff there is winning $\star$-strategy of size at most k in G. To this end, we first make several observations on G. First, there is a winning strategy in G since all predecessors of *bad* are Player A states that have other successors that bad. Second, Player A can never avoid the states of the form X_i nor C_j, i.e. for all strategy σ: $\mathcal{X} \cup \mathcal{C} \subseteq \mathsf{Reach}\left(G_\sigma, \text{init}^A\right)$. This entails that, in all *winning* $\star$-strategy $\hat{\sigma}$, we must have $\hat{\sigma}(v) \neq \star$ for all $v \in \mathcal{X} \cup \mathcal{C}$. Otherwise, if $\hat{\sigma}(v) = \star$ for some $v \in \mathcal{X} \cup \mathcal{C}$, there is at least one concretisation σ of $\hat{\sigma}$ s.t. $\sigma(v) = \text{bad}$, and thus bad is reachable in G_σ by the path $\text{init}^A, \text{init}^B, v, \text{bad}$, which contradicts the fact that $\hat{\sigma}$ is winning. Finally, consider a winning $\star$-strategy $\hat{\sigma}$. We have shown above that $\hat{\sigma}(X_i) \notin \{\star, \text{bad}\}$ for all $1 \leq i \leq m$. This implies that $\hat{\sigma}$ maps each X_i either to x^B_i, or to $\neg x^B_i$, and those nodes cannot be avoided by Player A, whatever concretisation of $\hat{\sigma}$ he plays. Using the same arguments as above, we conclude that $\hat{\sigma}(v)$ must be different from $\star$

for exactly m states in $\mathcal{L}^A$. More precisely, for all winning $\star$-strategies $\hat{\sigma}$, let $S_{\hat{\sigma}} = \{x_i^A \mid \exists 1 \leq i \leq m : \hat{\sigma}(X_i) = x_i^B\} \cup \{\neg x_i^A \mid \exists 1 \leq i \leq m : \hat{\sigma}(X_i) = \neg x_i^B\}$. Then, for all winning $\star$-strategies $\hat{\sigma}$, for all $v \in S_{\hat{\sigma}}$: $\hat{\sigma}(v) \neq \star$. Observe that $|S_{\hat{\sigma}}| = m$ for all winning $\star$-strategies $\hat{\sigma}$. We conclude that all winning $\star$-strategies $\hat{\sigma}$ is of size at least $k = 2 \times m + n$ in G.

Let us now conclude the proof. First consider a satisfying assignment $a : X \mapsto \{\mathbf{true}, \mathbf{false}\}$ for ϕ. Let $\hat{\sigma}$ be the $\star$-strategy defined as follows. For all $1 \leq i \leq m$: $\hat{\sigma}(X_i) = x_i^B$ if $a(x_i) = \mathbf{true}$ and $\hat{\sigma}(X_i) = \neg x_i^B$ otherwise. For all $1 \leq i \leq n$: pick a literal ℓ_i from C_i which is true under the assignment a (such a literal must exist since a makes ϕ true), and let $\hat{\sigma}(C_i) = \ell_i^B$. For all $v \in \{x_1^A, \neg x_1^A, \ldots, x_n^A, \neg x_n^A\}$, let $\hat{\sigma}(v) = \text{init}^B$ if $v = \hat{\sigma}(X_i)$ for some $1 \leq i \leq n$, and let $\hat{\sigma}(v) = \star$ otherwise. Finally, let $\hat{\sigma}(\text{init}^A) = \star$. It is easy to check that $\hat{\sigma}$ is indeed a *winning* $\star$-strategy of size exactly k.

Second, we assume a winning $\star$-strategy $\overline{\sigma}$ of size $\leq k$. By the above arguments, $\overline{\sigma}$ is of size exactly k and $\overline{\sigma}(v) \neq \star$ for all $v \in \mathcal{X} \cup \mathcal{C} \cup S_{\overline{\sigma}}$. Let us consider the assignment of the variables of ϕ that maps each variable x_i to **true** iff $\overline{\sigma}(X_i) = x_i^B$. To show that this assignment satisfies ϕ it is sufficient to show that $\overline{\sigma}(\mathcal{C}) \subseteq \overline{\sigma}(\mathcal{X})$, because this entails, by definition of G, that, under this assignment, each clause j contains at least one true literal (the one corresponding to $\overline{\sigma}(C_j)$). To establish that $\overline{\sigma}(\mathcal{C}) \subseteq \overline{\sigma}(\mathcal{X})$, we proceed by contradiction. If it is not the case, let $v \in \mathcal{C}$ be a state s.t. $\overline{\sigma}(v) \notin \overline{\sigma}(\mathcal{X})$. Assume $\overline{\sigma}(v) = \sim x_k^B$ for some k, and where $\sim$ can be $\neg$ or nothing. Then, by definition of G, the corresponding A state $\sim x_k^A$ is reachable in G_σ for all concretisation σ of $\overline{\sigma}$. Moreover, $\sim x_k^A \notin X \cup C \cup S_{\overline{\sigma}}$, hence $\overline{\sigma}(\sim x_k^A) = \star$, and there is at least one concretisation of $\overline{\sigma}$ that maps $\sim x_k^A$ to bad. Since $\sim x_k^A$ is reachable in this concretisation in particular, we conclude that $\overline{\sigma}$ is not winning. Contradiction. □

Thus, unless P=NP, there is no polynomial-time algorithm to compute a winning $\star$-strategy *of minimal size*. In most practical cases we are aware of, the situation is even worse, since the arena is not given explicitly. This is the case with the three problems we consider as applications (see Section 6), because they can be reduced to a safety games whose size is at least *exponential* in the size of the original problem instance.

4 Structured games and monotonic strategies

To mitigate the strong complexity identified in the previous section, we propose to follow the successful *antichain approach* [12,7,8]. In this line of research, the authors point out the fact that, in practical applications (like those we identify in Section 6), system states exhibit some inherent *structure*, which is formalised by a *simulation relation* and can be exploited to improve the practical running time of the algorithms. In the present paper, we rely on the notion of *turn-based alternating simulation*, to define heuristics to (i) improve the running time of the algorithms to solve safety games and (ii) obtain succinct representations of strategies. This notion is a straightforward adaptation, to turn-based games, of the alternating simulations from [2].

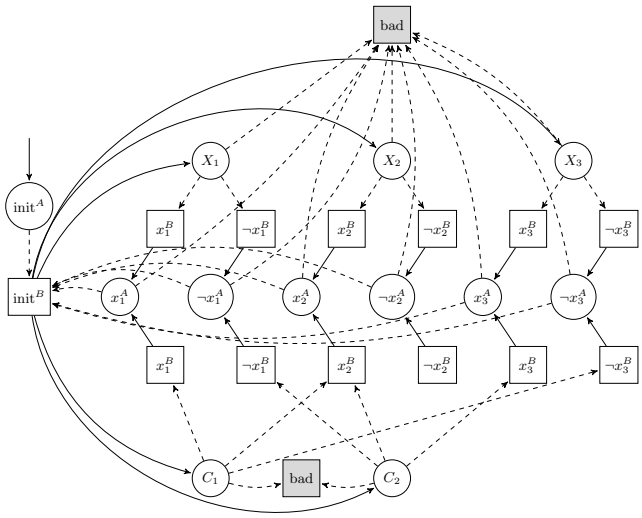

Fig. 2. Construction for the formula $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$: $k = 8$. State *bad*, and all the states of the form $(\neg)x_i^B$ have been duplicated for readability.

Turn-based alternating simulations Let $G = (V_A, V_B, E, I, \mathsf{Bad})$ be a finite safety game. A partial order $\trianglerighteq \subseteq V_A \times V_A \cup V_B \times V_B$ is a *turn-based alternating simulation relation for* G [2] (tba-simulation for short) iff for all v_1, v_2 s.t. $v_1 \trianglerighteq v_2$, either $v_1 \in \mathsf{Bad}$ or the three following conditions hold: (i) If $v_1 \in V_A$, then, for all $v_1' \in \mathsf{Succ}(v_1)$, there is $v_2' \in \mathsf{Succ}(v_2)$ s.t. $v_1' \trianglerighteq v_2'$; (ii) If $v_1 \in V_B$, then, for all $v_2' \in \mathsf{Succ}(v_2)$, there is $v_1' \in \mathsf{Succ}(v_1)$ s.t. $v_1' \trianglerighteq v_2'$; and (iii) $v_2 \in \mathsf{Bad}$ implies $v_1 \in \mathsf{Bad}$.

On the running example (Fig. 1), $\trianglerighteq_0$ is a tba-simulation relation. Indeed, as we are going to see in Section 6, a simulation relation in a game where player A has always the opportunity to perform the same moves is necessarily alternating.

Monotonic concretisations of $\star$-strategies Let us exploit the notion of tba-simulation to introduce a finer notion of concretisation of $\star$-strategies. Let $\hat{\sigma}$ be a $\star$-strategy. Then, a strategy σ is a $\trianglerighteq$-*concretisation* of $\hat{\sigma}$ iff for all $v \in V_A$:

$$v \in \mathsf{Supp}(\hat{\sigma}) \implies \sigma(v) = \hat{\sigma}(v)$$

and

$$\big(v \notin \mathsf{Supp}(\hat{\sigma}) \wedge v \in {\downarrow^{\trianglerighteq}}(\mathsf{Supp}(\hat{\sigma}))\big) \implies \exists \overline{v} \in \mathsf{Supp}(\hat{\sigma}) : \overline{v} \trianglerighteq v \wedge \sigma(\overline{v}) \trianglerighteq \sigma(v)$$

Intuitively, when $\hat{\sigma}(v) = \star$, but there is $v' \trianglerighteq v$ s.t. $\hat{\sigma}(v') \neq \star$, then, $\sigma(v)$ must mimic the strategy $\sigma(\overline{v})$ from some state $\overline{v}$ that covers v and s.t. $\hat{\sigma}(\overline{v}) \neq \star$. Then, we say that a $\star$-strategy is $\trianglerighteq$-*winning* if all its $\trianglerighteq$-concretisations are winning.

Because equality is a tba-simulation, the proof of Theorem 1 can be used to show that computing a $\trianglerighteq$-winning $\star$-strategy of size less than k is an NP-complete problem too. Nevertheless, $\trianglerighteq$-winning $\star$- strategies can be even more

compact than winning $\star$-strategy. For instance, on the running example, the smallest winning $\star$-strategy $\overline{\sigma}$ is of size 5: it is given in Fig. 1 (b) and highlighted by bold arrows in Fig. 1 (thus, $\overline{\sigma}(④) = \overline{\sigma}(⑦) = \star$). Yet, one can define a $\trianglerighteq_0$-winning $\star$-strategy $\hat{\sigma}$ of size 2 because states ⑤ and ⑥ simulate all the winning states of A. This $\star$-strategy[4] $\hat{\sigma}$ is the one given in Fig. 1 (c) and represented by the boldest arrows in Fig. 1. Observe that, while all $\trianglerighteq$-concretisations of $\hat{\sigma}$ are winning, it is not the case of all *concretisations* of $\hat{\sigma}$. For instance, there is one concretisation σ of $\hat{\sigma}$ s.t. $\sigma(⓪) = \boxed{2}$, but σ is not a $\trianglerighteq_0$-monotonic concretisation of $\hat{\sigma}$ (and is losing).

Obtaining $\trianglerighteq$-winning $\star$-strategies The previous example clearly shows the kind of $\trianglerighteq$-winning $\star$-strategies we want to achieve: $\star$-strategies $\hat{\sigma}$ s.t. $\mathsf{Supp}(\hat{\sigma})$ is the maximal antichain of the winning states. In Section 1, we introduce an efficient on-the-fly algorithm to compute such a $\star$-strategy. Its correctness is based on the fact that we can extract a $\trianglerighteq$-winning $\star$-strategy from any winning (plain) strategy, as shown by Proposition 1 hereunder. For all strategy σ, and all $V \subseteq V_A$, we let $\sigma|_V$ denote the $\star$-strategy $\hat{\sigma}$ s.t. $\hat{\sigma}(v) = \sigma(v)$ for all $v \in V$ and $\hat{\sigma}(v) = \star$ for all $v \notin V$. Then:

Proposition 1. *Let $G = (V_A, V_B, E, I, \mathsf{Bad})$ be a finite turn-based safety game and $\trianglerighteq$ be a tba-simulation relation for G. Let σ be a strategy in G, and let $\mathcal{S} \subseteq V_A$ be a set of A-states s.t.: (i) $(\mathcal{S} \cup \sigma(\mathcal{S})) \cap \mathsf{Bad} = \emptyset$; (ii) $I \in {\downarrow^{\trianglerighteq}}(\mathcal{S})$; and (iii) $\mathsf{succ}(\sigma(\mathcal{S})) \subseteq {\downarrow^{\trianglerighteq}}(\mathcal{S})$. Then, $\sigma|_{\mathcal{S}}$ is a $\trianglerighteq$-winning $\star$-strategy.*

Proof. Let τ be a $\trianglerighteq$-concretisation of $\sigma|_{\mathcal{S}}$ and let us show that τ is winning. Let us first show that all A-states reachable in G under strategy τ are covered by some state in $\mathcal{S}$, i.e. that $\mathsf{Reach}\,(G_\tau) \cap V_A \subseteq {\downarrow}(\mathcal{S})$. Let us consider $v \in \mathsf{Reach}\,(G_\tau) \cap V_A$, and let $v_0^A, v_1^B, v_1^A, \ldots, v_n^B, v_n^A$ be a path in in G_τ that reaches v, i.e., with $v_0^A = I$ and $v_n^A = v$. Let us prove, by induction on i that $v_i^A \in {\downarrow}(\mathcal{S})$ for all $0 \leq i \leq n$.
Base case $i = 0$: trivial by hypothesis (ii).
Inductive case $i = k > 0$: let us assume that $v_{k-1}^A \in {\downarrow}\,(\mathcal{S})$ and let us show that $v_k^A \in {\downarrow}(\mathcal{S})$. Since (v_{k-1}^A, v_k^B) is an edge in G_τ, $v_k^B = \tau(v_{k-1}^A)$. Since τ is a $\trianglerighteq$-concretisation of $\sigma|_{\mathcal{S}}$, there is $\overline{v} \in \mathcal{S}$ s.t. $\overline{v} \trianglerighteq v_{k-1}^A$ and $\sigma(\overline{v}) = \tau(\overline{v}) \trianglerighteq \tau(v_{k-1}^A) = v_k^B$. Thus, $\sigma(\overline{v}) \trianglerighteq v_k^B$. Since $\trianglerighteq$ is a tba-simulation, the successor v_k^A of v_k^B can be simulated by some successor $\hat{v}$ of $\sigma(\overline{v})$, i.e. $\hat{v} \trianglerighteq v_k^A$. By hypothesis (iii), $\hat{v} \in {\downarrow^{\trianglerighteq}}(\mathcal{S})$. Hence $v_k^A \in {\downarrow^{\trianglerighteq}}(\mathcal{S})$ too.

Next, let us expand that result by showing that all states reachable in G_τ are covered by some state in $\mathcal{S} \cup \tau(\mathcal{S})$, i.e. that $\mathsf{Reach}\,(G_\tau) \subseteq {\downarrow}\,(\mathcal{S} \cup \tau(\mathcal{S}))$. To do so, it is sufficient to show that each $v_B \in \mathsf{Reach}\,(G_\tau) \cap V_B \in {\downarrow}\,(\tau(\mathcal{S}))$. Since $v_B \in \mathsf{Reach}\,(G_\tau)$, there is $v_A \in \mathsf{Reach}\,(G_\tau) \cap V_A$ s.t. $(v_A, v_B) \in E$. Hence, $v_A \in {\downarrow}\,(\mathcal{S})$ by the arguments above. Thus, since τ is a $\trianglerighteq$-concretisation of $\sigma|_{\mathcal{S}}$, there is $\overline{v} \in \mathcal{S}$ s.t. $\overline{v} \trianglerighteq v_A$ and $\tau(v_A) \trianglerighteq \tau(v_B)$. Hence, $v_B \in {\downarrow^{\trianglerighteq}}\,(\tau(\mathcal{S}))$.

We conclude the proof by observing that that $\tau(\mathcal{S}) = \sigma(\mathcal{S})$, since τ is a $\trianglerighteq$-concretisation of $\sigma|_{\mathcal{S}}$. Hence, ${\downarrow}\,(\mathcal{S} \cup \tau(\mathcal{S})) = {\downarrow}\,(\mathcal{S} \cup \sigma(\mathcal{S}))$. Moreover, hypothesis (i) implies that ${\downarrow}\,(\mathcal{S} \cup \sigma(\mathcal{S})) \cap \mathsf{Bad} = \emptyset$, by definition of tba-simulation

[4] Actually, this strategy is winning for all initial number n of balls s.t. $n \mod 3 \neq 1$.

(compatible with Bad). Hence, since $\mathsf{Reach}\,(G_\tau) \subseteq \downarrow(\mathcal{S} \cup \tau(\mathcal{S})) = \downarrow(\mathcal{S} \cup \sigma(\mathcal{S}))$ by the arguments above, we conclude that $\mathsf{Reach}\,(G_\tau) \cap \mathsf{Bad} = \emptyset$, and thus, that τ is winning. □

This proposition allows us to identify families of sets of states on which $\star$-strategies can be defined. One of the sets that satisfies the conditions of Proposition 1 is the maximal antichain of reachable A-states, for a given winning strategy σ:

Theorem 2. *Let $G = (V_A, V_B, E, I, \mathsf{Bad})$ be a finite turn-based safety game, $\trianglerighteq$ be a tba-simulation relation for G. Let σ be a winning strategy and $\mathcal{WR}_\sigma$ be a maximal $\trianglerighteq$-antichain on $\mathsf{Reach}(G_\sigma) \cap V_A$, then the $\star$-strategy $\sigma|_{\mathcal{WR}_\sigma}$ is $\trianglerighteq$-winning.*

Proof. It is straightforward to verify that σ and $\mathcal{WR}_\sigma$ satisfy the three properties of Proposition 1. Indeed: (i) $\mathcal{WR}_\sigma \cup \sigma(\mathcal{WR}_\sigma) \subseteq \mathsf{Reach}(G_\sigma)$ by definition of $\mathcal{WR}_\sigma$ and $\mathsf{Reach}\,(G_\sigma) \cap \mathsf{Bad} = \emptyset$ because σ is winning. Hence $(\mathcal{WR}_\sigma \cup \sigma(\mathcal{WR}_\sigma)) \cap \mathsf{Bad} = \emptyset$. (ii) $I \in \mathsf{Reach}\,(G_\sigma) \cap V_A$ and $\mathsf{Reach}\,(G_\sigma) \cap V_A \subseteq \downarrow(\mathcal{WR}_\sigma)$ by definition, hence $I \in \downarrow(\mathcal{WR}_\sigma)$. (iii) $\mathsf{Succ}\,(\sigma)\,(\mathcal{WR}_\sigma)) \subseteq \mathsf{Reach}\,(G_\sigma)$ and $\mathsf{Reach}\,(G_\sigma) \subseteq \downarrow(\mathcal{WR}_\sigma)$ by definition, hence $\mathsf{Succ}\,(\sigma(\mathcal{WR}_\sigma)) \subseteq \downarrow(\mathcal{WR}_\sigma)$. □

5 Efficient computation of succinct winning strategies

The original OTFUR algorithm The OTFUR algorithm [6] is an efficient, on-the-fly algorithm to compute a winning strategy in a finite *reachability* game (it is thus easy to adapt it to solve safety games). We sketch the main ideas behind this algorithm, and refer the reader to [6] for a comprehensive description[5]. The intuition of the approach is to combine a forward exploration from the initial state with a backward propagation of the information when a losing state is found. During the forward exploration, newly discovered states are assumed winning until they are declared losing for sure. Whenever a losing state is identified (either because it is Bad, or because Bad is unavoidable from it), the information is back propagated to predecessors whose status could be affected by this information. A bookkeeping function Depend is used for that purpose: it associates, to each state v, a list $\mathsf{Depend}(v)$ of edges that need to be re-evaluated should v be declared losing. The main interest of this algorithm is that it works *on-the-fly* (thus, the arena does not need to be fully constructed before the analysis), and avoids, if possible, the entire traversal of the arena. In this section, we propose an optimized version of OTFUR for games equipped with tba-simulations. Before this, we prove that, when a safety game is equipped with a tba-simulation $\trianglerighteq$, then its set of winning states is $\trianglerighteq$-*downward closed*. This property will be important for the correctness of our algorithm.

We start by an auxiliary lemma that relates tba-simulations and computation of the attractor set:

[5] See Appendix A for the original algorithm.

Lemma 1. *Let G be a finite safety game, and let $\trianglerighteq$ be an tba-simulation for G. Then, for all $(v_1, v_2) \in \trianglerighteq$ and for all $i \in \mathbb{N}$: $v_2 \in \mathsf{Attr}_i$ implies $v_1 \in \mathsf{Attr}_i$.*

Proof. The proof is by induction on i.
Base case: $i = 0$ By definition, $\mathsf{Attr}_0 = \mathsf{Bad}$. However $v_2 \in \mathsf{Bad}$ and $v_1 \trianglerighteq v_2$ imply $v_1 \in \mathsf{Bad} = \mathsf{Attr}_0$, by definition of tba-simulations.
Inductive case: $i > 0$ Assume $v_2 \in \mathsf{Attr}_i$. There are two cases to study.

- In the case where $v_2 \in V_B$: first of all $v_1 \trianglerighteq v_2$ implies $v_1 \in V_B$. Moreover by definition of Attr_i, there exists $v_2' \in V_A$ such that $(v_2, v_2') \in E$ and $v_2' \in \mathsf{Attr}_{i-1}$. Then by definition of the tba-simulation relation, there exists $v_1' \in V_A$ such that $(v_1, v_1') \in E$ and $v_1' \trianglerighteq v_2'$. By induction hypothesis, $v_1' \in \mathsf{Attr}_{i-1}$ and hence by definition of Attr_i, $v_1 \in \mathsf{Attr}_i$.
- In the case where $v_2 \in V_A$: first of all $v_1 \trianglerighteq v_2$ implies $v_1 \in V_A$. Moreover by definition of Attr_i, for all $v_2' \in V_B$ such that $(v_2, v_2') \in E$, $v_2' \in \mathsf{Attr}_{i-1}$. On the other hand by definition of the tba-simulation relation, for all $v_1' \in V_B$ such that $(v_1, v_1') \in E$, then there exists $v_2'' \in V_B$ such that $(v_2, v_2'') \in E$ and $v_1' \trianglerighteq v_2''$. As a consequence $v_2'' \in \mathsf{Attr}_{i-1}$, and thus by induction hypothesis $v_1' \in \mathsf{Attr}_{i-1}$. Hence by definition of Attr_i, $v_1 \in \mathsf{Attr}_i$. □

Then:

Proposition 2. *Let G be a finite turn-based safety game, and let $\trianglerighteq$ be a tba-simulation for G. Then the set* Win *of winning states in G is downward closed for $\trianglerighteq$.*

Proof (Proposition 2). As a consequence of Lemma 1, for all $(v_1, v_2) \in \trianglerighteq$, $v_2 \in \mathsf{Attr}_{\mathsf{Bad}}$ implies $v_1 \in \mathsf{Attr}_{\mathsf{Bad}}$. Since Win is the complementary of $\mathsf{Attr}_{\mathsf{Bad}}$, $v_1 \in \mathsf{Win}$ implies $v_2 \in \mathsf{Win}$, that is Win is downwards closed for $\trianglerighteq$. □

Optimised OTFUR Let us discuss Algorithm 1, an optimised version of OTFUR for the construction of $\trianglerighteq$-winning $\star$-strategies in games with tba-simulations. Its high-level principle is the same than in the original OTFUR, i.e. forward exploration and backward propagation. At all times, it maintains several sets: (i) Waiting that stores edges waiting to be explored; (ii) Passed that stores nodes that have already been explored; and (iii) AntiLosing and AntiMaybe which represent, by means of antichains (see discussion below) a set of surely losing states and a set of possibly winning states respectively. The main **while** loop runs until either no more edges are waiting, or the initial state I is surely losing. An iteration of the loop first picks an edge $e = (v, v')$ from Waiting, and checks whether the exploration of this edge can be postponed (line 7–15, see discussion of the optimisations hereunder). Then, if v' has not been explored before (line 16), cannot be declared surely losing (line 18), and does not belong to Bad (line 19), it is explored (lines 23–28). When v' is found to be losing, e is put back in Waiting for back propagation (lines 21 or 30). The actual back-propagation is performed at lines 32–38 and triggered by an edge (v, v') s.t. $v' \in$ Passed. Let us highlight the three optimisations that prune the state space using a tba-simulation $\trianglerighteq$ on the game states:

Algorithm 1: The OTFUR optimized for games with a tba-simulation

```
   Data: A safety game G = (V_A, V_B, E, I, Bad)
 1 Passed := {I} ; Depend(I) := ∅ ;
 2 AntiMaybe := {I} ; AntiLosing := {} ;
 3 Waiting := {(I, v') | v' ∈ ⌊Succ(I)⌋} ;
 4 while Waiting ≠ ∅ ∧ I ∉↑ AntiLosing do
 5     e = (v, v') := pop(Waiting) ;
 6     if v ∉↑ AntiLosing then
 7         if v ∈↓ AntiMaybe \ AntiMaybe then
 8             choose v_m ∈ AntiMaybe s.t. v_m ⊵ v ;
 9             Depend[v_m] := Depend[v_m] ∪ {e} ;
10         else
11             if v' ∈↓ AntiMaybe then
12                 if v' ∉ AntiMaybe then
13                     choose v_m ∈ AntiMaybe s.t. v_m ⊵ v' ;
14                     Depend[v_m] := Depend[v_m] ∪ {e} ;
15             else
16                 if v' ∉ Passed then
17                     Passed := Passed ∪ {v'} ;
18                     if v' ∉↑ AntiLosing then
19                         if (v' ∈ Bad) then
20                             AntiLosing := ⌊AntiLosing ∪ {v'}⌋ ;
21                             Waiting := Waiting ∪ {e} ; // reevaluation of e
22                         else
23                             Depend[v'] := {(v, v')} ;
24                             AntiMaybe := ⌈AntiMaybe ∪ {v'}⌉ ;
25                             if v ∈ V_A then
26                                 Waiting := Waiting ∪ {(v', v'') | v' ∈ ⌊Succ(v')⌋} ;
27                             else
28                                 Waiting := Waiting ∪ {(v', v'') | v' ∈ ⌈Succ(v')⌉} ;
29                     else // reevaluation of e
30                         Waiting := Waiting ∪ {e} ;
31                 else // reevaluation
32                     Losing* := v ∈ V_A ∧ ⋀_{v''∈min(Succ(v))}(v'' ∈↑ AntiLosing)
                                ∨ v ∈ V_B ∧ ⋁_{v''∈max(Succ(v))}(v'' ∈↑ AntiLosing) ;
33                     if Losing* then
34                         AntiLosing := ⌊AntiLosing ∪ {v}⌋ ;
35                         AntiMaybe := ⌈Passed\ ↑(AntiLosing)⌉ ;
                           // back propagation
36                         Waiting := Waiting ∪ Depend[v] ;
37                     else
38                         if ¬Losing[v'] then Depend[v'] := Depend[v'] ∪ {e} ;
39 return I ∉↑ AntiLosing
```

1. By the properties of $\trianglerighteq$, we can explore only the $\trianglerighteq$-minimal (respectively $\trianglerighteq$-maximal) successors of each A (B) state (see lines 3, 26 and 28). We also consider maximal and minimal elements only when evaluating the status of a node in line 32.
2. By Proposition 2, the set of winning states in the game is downward-closed, hence the set of losing states is upward-closed, and we store the set of states that are losing for sure as an antichain $\mathsf{AntiLosing}$ of minimal losing states.
3. Symmetrically, the set of *possibly winning states* is stored as an antichain $\mathsf{AntiMaybe}$ of maximal states. This set allows to postpone, and potentially avoid, the exploration of some states: assume some edge (v, v') has been popped from $\mathsf{Waiting}$. Before exploring it, we first check whether either v or v' belongs to $\downarrow(\mathsf{AntiMaybe})$ (see lines 7 and 11). If yes, there is $v_m \in \mathsf{AntiMaybe}$ s.t. $v_m \trianglerighteq v$ (resp. $v_m \trianglerighteq v'$), and the exploration of v (v') can be postponed. We store the edge (v, v') that we were about to explore in $\mathsf{Depend}[v_m]$, so that, if v_m is eventually declared losing (see line 36), (v, v') will be re-scheduled for exploration. Thus, the algorithm can stop when all maximal A states have a successor that is covered by a non-losing one.

Observe that optimisations 1 and 2 rely only the upward closure of the losing states and were present in the antichain algorithm of [8]. Optimisation 3 is original and exploits more aggressively the notion of tba-simulation. It allows to keep at all times an antichain of potentially winning states, which is crucial to compute efficiently a winning $\star$-strategy. If, at the end of the execution, $I \notin \uparrow(\mathsf{AntiLosing})$, we can extract from $\mathsf{AntiMaybe}$ a winning $\star$-strategy $\hat{\sigma}_G$ as follows. For all $v \in \mathsf{AntiMaybe} \cap V_A$, we let $\hat{\sigma}_G(v) = v'$ such that $v' \in \mathsf{Succ}(v) \cap \downarrow(\mathsf{AntiMaybe})$. For all $v \in V_A \setminus \mathsf{AntiMaybe}$, we let $\hat{\sigma}_G(v) = \star$. Symmetrically, if $I \in \uparrow(\mathsf{AntiLosing})$, there is no winning strategy for A.

Correctness The correctness of Algorithm 1 is given by the following theorem.

Theorem 3. *When called on game G, Algorithm 1 always terminates. Upon termination, either $I \in \uparrow(\mathsf{AntiLosing})$ and there is no winning strategy for A in G, or $\hat{\sigma}_G$ is a $\trianglerighteq$-winning $\star$-strategy.*

We split the proof of Theorem 3 into two main propositions establishing respectively termination and soundness of the algorithm. The proof of soundness relies on auxiliary lemmata establishing invariants of the algorithm. In those proofs, we rely on the following notations. First of all, let us denote by $Name_i$ the state of the set $Name$ at the i-th iteration of the **while** in Algorithm 1. Let us define two notations: $\mathsf{StateWaiting}_i = \{v | \exists v', (v, v') \in \mathsf{Waiting}_i \text{ or } (v', v) \in \mathsf{Waiting}_i\}$, $\mathsf{Visited}_i = \{v | \exists j \leq i, v \in \mathsf{StateWaiting}_j\}$, and $S_i = \mathsf{AntiMaybe}_i \cup \mathsf{StateWaiting}_i$. In words, $\mathsf{StateWaiting}_i$ is the set of states which appear in $\mathsf{Waiting}$ at the i-th iteration of **while**, $\mathsf{Visited}_i$ is the set of the states which have appeared in $\mathsf{Waiting}$ at some iteration before the i-th one, and S_i is the set of the states which appear in $\mathsf{Waiting}_i$ or which belong to $\mathsf{AntiMaybe}_i$. Finally, we denote by $\mathsf{AntiMaybe}$, $\mathsf{Visited}$, $\mathsf{StateWaiting}$, $\mathsf{AntiLosing}$ and $\mathsf{Waiting}$ the state of those sets at the end of the execution of the algorithm.

Proposition 3 (Termination). *Algorithm 1 always terminates*

Proof. In order to prove the termination of Algorithm 1, we simply need to prove that the **while** loop cannot be iterated infinitely because each iteration of this loop takes finitely many steps. Let us prove it by contradiction. Let us assume that there is an infinite execution of Algorithm 1. Observe that for all indices $i < j$, $\mathsf{Passed}_i \subseteq \mathsf{Passed}_j$ and $\uparrow(\mathsf{AntiLosing})_i \subseteq \uparrow(\mathsf{AntiLosing})_j$. Hence, let K be s.t. for all $i \geq K$, $\mathsf{Passed}_i = \mathsf{Passed}_K$ and $\uparrow(\mathsf{AntiLosing})_i = \uparrow(\mathsf{AntiLosing})_K$, i.e. after K steps, Passed and AntiLosing stabilise and remain the same along the rest of the infinite run. Such a K necessarily exists because there are finitely many states in the arena. Now, we can easily check in the code that if Passed and $\uparrow(\mathsf{AntiLosing})$ are not modified in an iteration i (i.e., $\mathsf{Passed}_i = \mathsf{Passe}_{i-1}$ and $\uparrow(\mathsf{AntiLosing_i}) = \uparrow(\mathsf{AntiLosing_{i-1}})$) then Waiting decreases strictly, i.e. $\mathsf{Waiting}_i \subseteq \mathsf{Waiting}_{i-1}$. Since at step K, $\mathsf{Waiting}_K$ is necessarily finite, and since AntiLosing and Passed stay constant from step K, there is a step $K' \geq K$ s.t. $\mathsf{Waiting}_{K'} = \emptyset$. However, this implies that the algorithms stops at step K'. Contradiction. □

Let us now turn our attention to soundness. We first establish several invariants of the algorithm:

Lemma 2. *Algorithm 1 admits the three following loop-invariants:*

$$\mathsf{Inv}_i^1 : \mathsf{Visited}_i \setminus \mathsf{StateWaiting}_i \subseteq \downarrow(\mathsf{AntiMaybe})_i \cup \uparrow(\mathsf{AntiLosing})_i$$

$$\mathsf{Inv}_i^2 : \forall v \in \downarrow(\mathsf{AntiMaybe})_i \cap V_A, \exists v' \in \mathsf{Succ}(v) : v' \in \downarrow(\mathsf{AntiMaybe})_i \vee (v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})_i] \cup \mathsf{Waiting}_i$$

$$\mathsf{Inv}_i^3 : \forall v \in \downarrow(\mathsf{AntiMaybe})_i \cap V_B, \forall v' \in \mathsf{Succ}(v) : v' \in \downarrow(\mathsf{AntiMaybe})_i \vee (v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})_i] \cup \mathsf{Waiting}_i$$

Proof. We start by observing that the sets $\uparrow(\mathsf{AntiLosing})$ and Visited increase monotonically along the execution of the algorithm. Then let us prove each invariant separatly

(Inv_i^1) At $i = 0$, all the states in Visited are in StateWaiting, hence the initialization is trivial.
Let us now assume that Inv_i^1 is true for a given i and let us prove that Inv_{i+1}^1 is also true. Observe that the definition of $\mathsf{Visited}_i$ depends only on StateWaiting: $\mathsf{Visited}_i = \cup_{j \leq i} \mathsf{StateWaiting}_j$. This implies in particular that $\mathsf{StateWaiting}_i \subseteq \mathsf{Visited}_i$ for all i. Thus, we only need to consider the modification of StateWaiting during one iteration to prove this invariant. At each iteration of the loop, one edge is taken from Waiting, and several edges are potentially added. Thus, the sets $\mathsf{StateWaiting}_{i+1} \setminus \mathsf{StateWaiting}_i$ and $\mathsf{StateWaiting}_i \setminus \mathsf{StateWaiting}_{i+1}$ are both potentially non-empty. This discussion allows to conclude that, for all states $v \in \mathsf{Visited}_{i+1} \setminus \mathsf{StateWaiting}_{i+1}$, we only need to consider two cases: (i) either $v \in \mathsf{Visited}_i \setminus \mathsf{StateWaiting}_i$; (ii) or $v \in \mathsf{StateWaiting}_i \setminus \mathsf{StateWaiting}_{i+1}$ since by definition, nodes in $\mathsf{StateWaiting}_{i+1} \setminus \mathsf{StateWaiting}_i$ are not in $\mathsf{Visited}_{i+1} \setminus \mathsf{StateWaiting}_{i+1}$.

1. Let v be in $\left(\mathsf{Visited}_{i+1} \setminus \mathsf{StateWaiting}_{i+1}\right) \cap \left(\mathsf{Visited}_i \setminus \mathsf{StateWaiting}_i\right)$. Since $v \in \mathsf{Visited}_i \setminus \mathsf{StateWaiting}_i$, v is in $\downarrow(\mathsf{AntiMaybe})_i \cup \uparrow(\mathsf{AntiLosing})_i$, by induction hypothesis.
   (a) If $v \in \uparrow(\mathsf{AntiLosing})_i$, then $v \in \uparrow(\mathsf{AntiLosing})_{i+1}$ (see the begining of the proof), hence, $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow(\mathsf{AntiLosing})_{i+1}$, and v respects the invariant.
   (b) Otherwise, $v \in \downarrow(\mathsf{AntiMaybe})_i$. Then, either $v \in \downarrow(\mathsf{AntiMaybe})_{i+1}$ (i.e., $\downarrow(\mathsf{AntiMaybe})$ has not decreased), which implies $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow(\mathsf{AntiLosing})_{i+1}$, and v respects the invariant. Or $v \notin \downarrow(\mathsf{AntiMaybe})_{i+1}$, i.e. $\mathsf{AntiMaybe}$ has decreased. This can occur only in line 35, but, in this case, all states that are removed from $\downarrow(\mathsf{AntiMaybe})$ are inserted in $\mathsf{StateWaiting}_{i+1}$ (actually, edges containing those states are inserted in $\mathsf{Waiting}_{i+1}$ in line 36), which contradicts our hypothesis that $v \in \mathsf{Visited}_{i+1} \setminus \mathsf{StateWaiting}_{i+1}$.
2. Otherwise, let v be in $v \in \mathsf{StateWaiting}_i \setminus \mathsf{StateWaiting}_{i+1}$. This can occur only because an edge (v_1, v_2) has been popped in line 5, with either $v = v_1$ or $v = v_2$. We consider those two cases separately.
   (a) If $v = v_1$, then, v is necessarily in Passed because lines 26 and 28 are the only lines where new edges are built and, if we execute one of these lines, adding an edge of the form (v_1, v_2) implies that v_1 has been added in Passed on line 17. One can also check that, in this conditional, v_1 is either added to $\mathsf{AntiMaybe}$ or to $\mathsf{AntiLosing}$. Thus, when a state is in Passed, then it will always be in $\downarrow(\mathsf{AntiMaybe}) \cup \uparrow(\mathsf{AntiLosing})$. Hence v belongs to $\downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow(\mathsf{AntiLosing})_{i+1}$.
   (b) Otherwise, if $v = v_2$, then either $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow(\mathsf{AntiLosing})_{i+1}$, or it is not already passed and the conditional on line 16 is satisfied. As a consequence, at the end of the iteration, $v \in \mathsf{AntiMaybe}_{i+1}$ or $v \in \mathsf{AntiLosing}_{i+1}$.

(Inv_i^2) At $i = 0$, the only state in $\mathsf{AntiMaybe}$ is I and all the edges of the form (I, v) are in $\mathsf{Waiting}$, hence the initialization is trivial.
Let us assume that Inv_i^2 is true for a given i and let us prove that Inv_{i+1}^2 is also true. To do so, we have to inspect all the cases which could make Inv_{i+1}^2 false when Inv_i^2 is true. For the sake of clarity, we use the ✓symbol to mean that a case is closed.

1. Let us assume that there exist $v \in \downarrow(\mathsf{AntiMaybe})_i$ and $v' \in \mathsf{Succ}(v)$ such that $v' \in \downarrow(\mathsf{AntiMaybe})_i \setminus \downarrow(\mathsf{AntiMaybe})_{i+1}$. The strict inclusion of $\downarrow(\mathsf{AntiMaybe})_{i+1}$ in $\downarrow(\mathsf{AntiMaybe})_i$ implies that line 35 has been executed during the i-th iteration of the loop. Then, either v' is the state which is put in $\mathsf{AntiLosing}_{i+1}$ on line 34 and $(v, v') \in \mathsf{Depend}_i[v']$ (see line 36) and thus $(v, v') \in \mathsf{Waiting}_{i+1}$ (✓), or $v' \in \mathsf{Passed}$ and by construction on line 35, v' necessarily belongs to $\downarrow(\mathsf{AntiMaybe})_{i+1}$ which contradicts our hypothesis (✓), or $v' \notin \mathsf{Passed}$ and there are two further possible cases. Indeed, if $v' \notin \mathsf{Passed}$, either $(v, v') \in \mathsf{Waiting}_i$ (✓), or the exploration of edge (v, v') has been postponed (lines 7–14) because there was a $\trianglerighteq$-greater state than v' in $\mathsf{AntiMaybe}$. In this latter case, either there is

again a $\trianglerighteq$-greater state than v' in $\mathsf{AntiMaybe}$ (✓), or (v, v') necessarily belongs to $\mathsf{Depend}_i[w]$ where w is the state put in $\mathsf{AntiLosing}_{i+1}$ (✓).

2. Let us assume that there exist $v \in \downarrow(\mathsf{AntiMaybe})_i$ and $v' \in \mathsf{Succ}(v)$ such that $(v, v') \in \mathsf{Waiting}_i \setminus \mathsf{Waiting}_{i+1}$. This implies that (v, v') has been popped (line 5) at the begining of the $i+1$th iteration, and there are three possible cases.
   (a) $(v, v') \in \mathit{Depend}_{i+1}[\downarrow(\mathsf{AntiMaybe})_{i+1}]$ (line 9 or 14) (✓)
   (b) $v' \in \mathsf{Passed}_{i+1} \setminus \mathsf{Passed}_i$ (line 17). Then, either (lines 20–21) $v' \in \uparrow(\mathsf{AntiLosing})_{i+1}$ and $(v, v') \in \mathsf{Waiting}_{i+1}$ which contradicts our hypothesis (✓), or (line 24) $v' \in \downarrow(\mathsf{AntiMaybe})_{i+1}$ (✓).
   (c) (v, v') goes in the "reevaluation" part. Then, either there is no successor of v outside of $\uparrow(\mathsf{AntiLosing})_i$ hence $v \notin \mathsf{AntiMaybe}_{i+1}$ (✓), or there exists a successor w of v which is not in $\uparrow(\mathsf{AntiLosing})_i$. In this latter case, either (v, w) has not already been treated and $(v, v') \in \mathsf{Waiting}_{i+1}$ (✓), or $w \in \downarrow(\mathsf{AntiMaybe})_{i+1}$ (✓).
3. Let us assume that there exist $v \in \downarrow(\mathsf{AntiMaybe})_i$ and $v' \in \mathsf{Succ}(v)$ such that $(v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})_i] \setminus \mathsf{Depend}_{i+1}[\downarrow(\mathsf{AntiMaybe})_{i+1}]$. This implies that a state w of $\mathsf{AntiMaybe}_i$ is found losing and added to $\mathsf{AntiLosing}_{i+1}$ (line 34). One can assume that $w \neq v'$ because the case $w = v'$ is treated in case 1 above. Then, since $(v, v') \notin \mathsf{Depend}_{i+1}[\downarrow(\mathsf{AntiMaybe})_{i+1}]$, $(v, v') \in \mathsf{Depend}_i[w]$ because, w is necessarily the only state in Passed_i which is in $\downarrow(\mathsf{AntiMaybe})_i \setminus \downarrow(\mathsf{AntiMaybe})_{i+1}$ (this holds since $\mathsf{AntiMaybe}_{i+1} = \lfloor \mathsf{Passed}_{i+1} \setminus \mathsf{Losing}_{i+1} \rfloor$, and w is the only state identified as losing during iteration $i+1$), and to have $\mathsf{Depend}_i[w'] \neq \emptyset$, it is necessary that w' is in Passed (✓).
4. Let us assume that there exist $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \setminus \downarrow(\mathsf{AntiMaybe})_i$. Then, line 26 or line 28 is executed and for all $v' \in \lceil \mathsf{Succ}(v) \rceil$: $(v, v') \in \mathsf{Waiting}_{i+1}$.

(Inv_i^2) The proof of (3) can be done with the disjunction of cases of the proof of (2). The only difference is for the case 2c. Indeed, the case is simpler because when (v, v') goes in the "reevaluation" part, it is not necessary to consider other successors, because v must have only non-losing successors. Then, either there is a successor of v in $\uparrow(\mathsf{AntiLosing})_i$ hence $v \notin \downarrow(\mathsf{AntiMaybe})_{i+1}$ (✓), or v' is in $\mathsf{Passed}_{i+1} \setminus \uparrow(\mathsf{AntiLosing})_i$ and thus belongs to $\downarrow(\mathsf{AntiMaybe})_{i+1}$ (✓). □

Lemma 3. *Algorithm 1 admits the following loop-invariant:*

$$\mathsf{Inv}_i^4 : \mathsf{AntiLosing}_i \subseteq \mathsf{Losing}$$

Proof. At $i = 0$, the $\mathsf{AntiLosing}$ is , hence the initialization is trivial.

Let us assume that Inv_i^4 is true for a given i and let us prove that Inv_{i+1}^4 is also true. We simply have to check that states in $\uparrow(\mathsf{AntiLosing})$ at the i-th iteration are losing.

There are two lines where states are added to $\uparrow(\mathsf{AntiLosing})$: lines 20 and 34.

– (line 20) A state v' can be added in $\mathsf{AntiLosing}$ on line 20. In this case, the conditional on line 19 ensures that the state is in Bad, hence v' is in Losing.
– (line 34) A state v can be added in $\mathsf{AntiLosing}$ on line 34. In this case, the definition of $Losing^*$ and the conditional on line 33 ensures that v is in Losing. Indeed, if $v \in V_A$ then all its minimal successors are in $\uparrow(\mathsf{AntiLosing})_i$, hence all its successors are losing by induction assumption and thus v is in Losing. Otherwise, $v \in V_B$ and it has a successor in $\uparrow(\mathsf{AntiLosing})_i$, hence one of its successors is losing by induction assumption and thus v is in Losing. □

Lemma 4. *Algorithm 1 admits the following loop-invariant:*

$$\mathsf{Inv}_i^5 : \forall \overline{v}, \forall (v, v') \in \mathsf{Depend}_i[\overline{v}] : \overline{v} \trianglerighteq v' \text{ or } (\overline{v} \trianglerighteq v \text{ and } \overline{v} \neq v)$$

Proof. The invariant is easily established by checking lines 9, 14 and 23, which are the only lines where an edge is added to Depend. □

We are now ready to prove soundness of the algorithm:

Proposition 4 (Soundness). *When Algorithm 1 terminates, either $I \in \uparrow(\mathsf{AntiLosing})$ and there is no winning strategy for A in this game, or σ is a $\trianglerighteq$-winning $\star$-strategy.*

Proof. We first consider the case where Algorithm 1 ends with $I \notin \uparrow(\mathsf{AntiLosing})$. In particular, at the end of the execution, $\mathsf{Waiting}$ is empty.

Let us first show that $\hat{\sigma}_G$ is well-defined, i.e. for all $v \in \mathsf{AntiMaybe} \cap V_A$, there is $v' \in \mathsf{Succ}(v)$ s.t. $v' \in \downarrow(\mathsf{AntiMaybe})$. This stems from Inv_i^1, Inv_i^2 and Inv_i^5. Indeed, at the end of the execution, $\mathsf{Waiting}$ is empty, hence $\mathsf{StateWaiting}$ is empty. Thus, Inv_i^2 entails that all $v \in \mathsf{AntiMaybe} \cap V_A$ have a successor v' s.t. either $v' \in \downarrow(\mathsf{AntiMaybe})$ or $(v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})]$. In the former case, the property is established. In the latter case ($(v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})]$), let $\overline{v}$ be a node from $\downarrow(\mathsf{AntiMaybe})$ s.t. $(v, v') \in \mathsf{Depend}_i(\overline{v})$. By Inv_i^5, either $\overline{v} \trianglerighteq v'$, or $\overline{v} \trianglerighteq v$ and $\overline{v} \neq v$. We first observe that the case '$\overline{v} \trianglerighteq v$ and $\overline{v} \neq v$' is not possible because $v \in \mathsf{AntiMaybe}$ by hypothesis, and $\overline{v} \in \downarrow(\mathsf{AntiMaybe})$. Thus, $\overline{v} \trianglerighteq v'$, which implies that $v' \in \downarrow(\mathsf{AntiMaybe})$ since $\overline{v} \in \downarrow(\mathsf{AntiMaybe})$. Thus, $\hat{\sigma}_G$ is well-defined.

We conclude the proof by invoking Proposition 1. To be able to apply this proposition, we need a strategy σ. We let σ be any concretisation of $\hat{\sigma}_G$, and $\mathcal{S} = \mathsf{AntiMaybe} \cap V_A$. The choice of the concretisation of $\hat{\sigma}_G$ does not matter, because the hypothesis required by Proposition 1 are properties of $\sigma(v)$ for states $v \in \mathcal{S}$ only, and $\mathcal{S} = \mathsf{AntiMaybe} \cap V_A$ is exactly the support of $\hat{\sigma}_G$. Let us show that σ respects the three hypothesis of Proposition 1, i.e. that: (i) $(\mathsf{AntiMaybe} \cap V_A \cup \hat{\sigma}_G(\mathsf{AntiMaybe} \cap V_A)) \cap \mathsf{Bad} = \emptyset$; (ii) $I \in \downarrow^{\trianglerighteq}(\mathsf{AntiMaybe} \cap V_A)$; and (iii) $\mathsf{succ}(\hat{\sigma}_G(\mathsf{AntiMaybe} \cap V_A)) \subseteq \downarrow^{\trianglerighteq}(\mathsf{AntiMaybe} \cap V_A)$.

(i) By definition of alternating simulation, Bad is upward closed ($\uparrow(\mathsf{Bad}) = \mathsf{Bad}$). No bad state can be added to $\mathsf{AntiMaybe}$ during the execution, because of the conditional in line 19 which is false when a state is added to $\mathsf{AntiMaybe}$ (line 24). Moreover by definition of $\hat{\sigma}_G$, $\hat{\sigma}_G(\mathsf{AntiMaybe} \cap V_A) \subseteq \downarrow(\mathsf{AntiMaybe})$, hence (i) is satisfied.

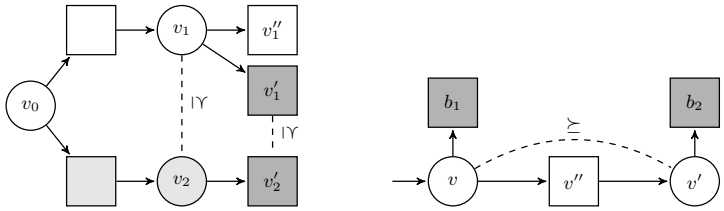

Fig. 3. A simulation and the downward closure are not sufficient to apply Algorithm 1.

(ii) By assumption, $I \notin \uparrow(\mathsf{AntiLosing})$. As a consequence of Inv_i^1, I belongs to $\downarrow(\mathsf{AntiMaybe})$, hence (ii) is satisfied.

(iii) By definition of $\hat{\sigma}_G$, $\hat{\sigma}_G(\mathsf{AntiMaybe} \cap V_A) \subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_B$. Now, by Inv_i^3, at the end of the execution, $\mathsf{Succ}(\downarrow(\mathsf{AntiMaybe}) \cap V_B) \subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_A$. Indeed, at the end of the execution, $\mathsf{StateWaiting}$ is empty and $(v, v') \in \mathsf{Depend}[\downarrow(\mathsf{AntiMaybe})]$ implies, by Inv_i^5, that either $v' \in \downarrow(\mathsf{AntiMaybe})$, or $v \in \downarrow(\mathsf{AntiMaybe})$ which means that there is a state w s.t. $w \trianglerighteq v$ in $\mathsf{AntiMaybe}$. In this latter case, by definition of the tba-simulation, there exists $\overline{v} \in \mathsf{Succ}(v)$ such that $\hat{\sigma}_G(w) \trianglerighteq \overline{v}$, therefore $\overline{v} \in \downarrow(\mathsf{AntiMaybe})$. Hence (iii) is satisfied.

Hence, by Proposition 1, the $\star$-strategy $\sigma|_{\mathcal{S}}$ (with $\mathcal{S} = \mathsf{AntiMaybe} \cap V_A$) is a $\trianglerighteq$-winning $\star$-strategy. It is easy to check that $\sigma|_{\mathcal{S}} = \hat{\sigma}_G$, by definition of σ.

We conclude the proof by considering the case where Algorithm 1 ends with $I \in \uparrow(\mathsf{AntiLosing})$. By Inv_i^4, there is no winning strategy for A in the game. □

Why simulations are not sufficient Let us exhibit two examples of games equipped with a simulation $\succeq$ which is not a tba-simulation, to show why tba-simulations are crucial for our optimisations. In Fig. 3 (left), $\mathsf{Bad} = \{v_1', v_2'\}$, and the set of winning states is not $\succeq$-downward closed (gray states are losing). In the game of Fig. 3 (right), $\mathsf{Bad} = \{b_1, b_2\}$ and Algorithm 1 does not develop the successors of v' (because $v \succeq v'$, and $v \in \mathsf{AntiMaybe}$ when first reaching v'). Instead, it computes a purportedly winning $\star$-strategy $\hat{\sigma}_G$ s.t. $\hat{\sigma}_G(v) = v''$ and $\hat{\sigma}_G(v') = \star$. Clearly this $\star$-strategy is not $\succeq$-winning (actually, there is no winning strategy in this game).

6 Applications

To show the relevance of our approach, let us briefly explain how to apply it to three problems that can be reduced to a safety game: LTL realisability, real-time scheduler synthesis and determinisation of timed automata. These three problems share the following characteristics, that make our technique particularly appealing: (i) they have practical applications where an efficient implementation of the winning strategy is crucial ; (ii) the arena of the safety game is not given explicitly and is at least exponential in the size of the problem instance; and (iii) they admit natural tba-simulations.

A-deterministic and $\trianglerighteq$-monotonic games To show that a *tba-simulation* exists in a safety game for which a *simulation relation* $\trianglerighteq$ is already known, we rely on the notions of *A-determinism* and $\trianglerighteq$-*monotonicity* of safety games. Intuitively, this means that player A can always chose to play *the same set of actions* in each of its states, and that playing the same action a in two states $v_1 \trianglerighteq v_2$ yields two states v_1' and v_2' with $v_1' \trianglerighteq v_2'$ [6]. Formally, let $G = (V_A, V_B, E, I, \mathsf{Bad})$ be a finite turn-based safety game and Σ a finite alphabet. A *labeling* of G is a function $\mathsf{lab} : E \to \Sigma$. For all states $v \in V_A \cup V_B$, and all $a \in \Sigma$, we let $\mathsf{Succ}_a(v) = \{v' \mid (v, v') \in E \wedge \mathsf{lab}(v, v') = a\}$ be the set of a-labeled successors of v. Then, (G, lab) is *A-deterministic* iff there is a set of actions $\Sigma_A \subseteq \Sigma$ s.t. for all $v \in V_A$: (i) $|\mathsf{Succ}_a(v)| = 1$ for all $a \in \Sigma_A$ and (ii) $|\mathsf{Succ}_a(v)| = 0$ for all $a \notin \Sigma_A$. Moreover, a labeling lab is $\trianglerighteq$-*monotonic* (where $\trianglerighteq$ is a simulation relation on the states of G) iff for all $v_1, v_2 \in V_A \cup V_B$ such that $v_1 \trianglerighteq v_2$, for all $a \in \Sigma$, for all $v_2' \in \mathsf{Succ}_a(v_2)$: there is $v_1' \in \mathsf{Succ}_a(v_1)$ s.t. $v_1' \trianglerighteq v_2'$. Then:

Theorem 4. *Let $G = (V_A, V_B, E, I, \mathsf{Bad})$ be a finite turn-based safety game, let $\trianglerighteq$ be a simulation relation on G and let* lab *be a* $\trianglerighteq$-monotonic *labeling of G. If (G, lab) is* A-deterministic*, then $\trianglerighteq$ is a tba-simulation relation.*

Proof. Since $\trianglerighteq$ is a simulation relation, we only need to prove that for all $v_1 \in V_A$, for all v_2 s.t. $v_1 \trianglerighteq v_2$, for all $v_1' \in \mathsf{Succ}(v_1)$, there is $v_2' \in V_B$ s.t. $v_2' \in \mathsf{Succ}(v_2)$ and $v_1' \trianglerighteq v_2'$. Let $v_1, v_2 \in V_A$ be s.t. $v_1 \trianglerighteq v_2$, and let v_1' be a state from $\mathsf{Succ}(v_1)$. Since G is A-deterministic, there is $v_2' \in \mathsf{Succ}(v_2)$ s.t. $\mathsf{lab}(v_1, v_1') = \mathsf{lab}(v_2, v_2')$. Since lab is $\trianglerighteq$-*monotonic*, we also have $v_2 \trianglerighteq v_2'$. □

Thus, when a safety game G is labeled, A-deterministic, $\trianglerighteq$-monotonic and equipped with a simulation relation $\trianglerighteq$ that can be computed directly on the description of the states, our approach can be applied out-of-the-box. In this case, the algorithm of Section 5 yields, if it exists, a winning $\star$-strategy $\hat{\sigma}_G$, that can be described by the set of pairs (v, a) for all v in the support of $\hat{\sigma}_G$ (the maximal antichain of winning reachable states), and where a is the label of the edge $(v, \hat{\sigma}_G(v))$. That is, we can store the *action* one needs to play from v instead of the *successor* $\hat{\sigma}_G(v)$. Observe that no information needs to be stored for the partial order $\trianglerighteq$ that can be directly computed on the description of the states. Then, a controller implementing $\hat{\sigma}_G$ works as follows: when the system state is v, the controller looks for a pair $(\overline{v}, a)$ in the strategy description with $\overline{v} \trianglerighteq v$. It executes action a from the current state v, which is possible by A-determinism, and respects the definition of $\trianglerighteq$-concretisation by $\trianglerighteq$-monotonicity. Finding $\overline{v}$ should be efficient, because we expect the antichain of winning reachable states to be compact. Let briefly explain why this technique applies to the three cases mentioned above.

LTL realisability LTL [10] is a popular logic to express properties of computer systems. An LTL formula defines a set of traces, i.e. infinite sequences of valuations of atomic propositions. In the LTL realisability problem [11], the set

[6] For example, in the urn-filling game (Fig. 1), Player A can always choose between taking 1 or 2 balls, from all states where at least 2 balls are left.

of atomic propositions is partitioned into *controllable* and *uncontrollable* ones. The controller and the environment are two players that compete in a game, where, at each turn, they fix the valuations of the atomic propositions they own, thereby building a trace. The play is winning for the controller iff the trace satisfies a given LTL formula. A formula is *realisable* iff the controller has a winning strategy in the game.

In [8], Filiot, Jin and Raskin reduce the realisability problem to a safety game, whose states are vectors of bounded natural numbers. They consider the partial order $\succeq$ on the states of the game, defined as $v \succeq v'$ iff $v[i] \geq v'[i]$ for all coordinates i. They show it is a *simulation relation* and rely on it to define an efficient antichain algorithm (based on the OTFUR algorithm) to solve the class of safety games they obtain from the realisability problem.

As a matter of fact, our technique generalises these results. Theorem 4 can be invoked to show that $\succeq$ is a *tba-simulation*. Hence, Algorithm 1 can be use to solve LTL realisability. As already explained, the antichain algorithm of [8] contains two of the three optimisations that are present in Algorithm 1 (see Section 5). Our results thus provide a general theory to explain the excellent performance of the technique of [8], and have the potential to further improve it.

Multiprocessor real-time scheduler synthesis We consider the problem of computing a correct scheduler for a given set of *sporadic real-time tasks*. A sporadic task (C, T, D) is a process that repeatedly creates *jobs*, s.t. each job creation (also called *request*) occurs at least T time units after the previous one. Each job models a computational payload. It needs at most C units of CPU time to complete, and must obtain them within a certain time frame of length D starting from the request (otherwise the job *misses* its deadline). We assume the tasks run on a platform of m identical CPUs. A scheduler is a function that assigns, at all times, jobs to available CPUs. A scheduler is *correct* iff it ensures that no job ever misses a deadline, whatever the sequence of requests.

This problem can be reduced to a safety game [4] where the two players are the scheduler and the coalition of the tasks respectively. In this setting, a *winning* strategy for Player A is a *correct* scheduler. In practice, this approach is limited by the size of the *game arena*, which is, in general, exponentially larger than the description of the set of tasks [4]. Nevertheless, we can rely on Theorem 4 to show that the game admits a tba-simulation relation. Indeed the simulation relation $\succeq$ introduced in [9] (to solve a related real-time scheduling problem using antichain techniques) naturally induces a simulation relation on the set of states of the game. An A-deterministic and $\succeq$-monotonic labeling is obtained if we label moves of the environment by the set of tasks producing a request, and the scheduler moves by a total order on all the tasks, which is used as a priority function determining which tasks are scheduled for running.

Determinisation of timed automata Timed automata (TA for short) [1] are a well-established model for real-time systems. TAs extend finite automata with clocks, that are real-valued variables evolving at the same rate, and that can be

constrained by guards on transitions. Fig. 4 presents a timed automaton with one clock x, locations ℓ_0, ℓ_1 and ℓ_2 and the alphabet $\{a, b\}$. A TA naturally defines a *timed language* and two timed automata are said to be *equivalent* if they admit the same language.

A TA is deterministic if from every state, at most one transition is firable for each action. The determinisation of a TA $\mathcal{A}$ is the construction of a deterministic TA equivalent to $\mathcal{A}$ and is a crucial operation for several problems such as test generation, fault diagnosis or more generally all the problems closed to the complement operation. Unfortunately, TAs are not determinisable in general [1]. Fig. 4 illustrates the difficulty: from location ℓ_0, there are two edges with action a and guard $0 < x < 1$, but x is reset on only one of these edges. Hence, a deterministic version of this TA should have at least two clocks to keep track of the two possible clock values on these two branches. Based on this idea, one can build a TA (with loops) for which the number of clock values that must be tracked simultaneously cannot be bounded. Furthermore, checking whether a given TA admits a deterministic version is an undecidable problem [1]. As a consequence, only partial algorithms exist for determinisation.

So far, the most general of those techniques has been introduced in [3] and consists in turning a TA $\mathcal{A}$ into a safety game $G_{\mathcal{A},(Y,M)}$ (parametrised by a set of clocks Y and a maximal constant M). Then, a deterministic TA over-approximating $\mathcal{A}$ (with set of clocks Y and maximal constant M), can be extracted from any Player A strategy. If the strategy is winning, then the approximation is an *exact* determinisation. The idea of the construction is that Player A actions consists in choosing a good reset policy for the clocks to avoid states in which the deterministic TA could over-approximate $\mathcal{A}$. States of the game can be seen as pairs $((S, S_\top), r)$ where S is a set of configurations corresponding to an approximate state estimate; $S_\top \subseteq S$ are the configurations corresponding to the state estimate which is surely not approximated; and r is a set of valuations of Y. Bad states which have to be avoided are states where $S_\top$ is empty. Then, a tba-simulation $\trianglerighteq_{\mathsf{det}}$ can naturally be defined on this game:

Lemma 5. *Let $\mathcal{A}$ be a timed automaton, Y be a set of clocks and M be a maximal constant for guards. Then $G_{\mathcal{A},(Y,M)}$ admits a tba-simulation relation $\trianglerighteq_{\mathsf{det}}$ defined as follows: $((S, S_\top), r) \trianglerighteq_{\mathsf{det}} ((S', S'_\top), r')$ iff $r = r'$, $S \supseteq S'$ and $S_\top \subseteq S_\top$.*

Proof. First of all, $\trianglerighteq_{\mathsf{set}}$ is clearly a partial order. It is thus sufficient to prove that this partial order is a simulation relation. To do so, we simply prove that moves of Player A and Player B preserve the partial order. Let $v^1 = ((S^1, S^1_\top), r^1)$ and $v^2 = ((S^2, S^2_\top), r^2)$ such that $v_1 \trianglerighteq_{\mathsf{set}} v_2$.

- **Moves of Player** A are resets of clocks. Let $Y' \subseteq Y$ a set of clocks that A can reset. Then, the Y'-successors of v^1 and v^2 are respectively $v'^1 = ((S'^1, S'^1_\top), r'^1)$ and $v'^2 = ((S'^2, S'^2_\top), r'^2)$, where for $i = 1$ or 2, $r'^i = r^i_{[Y' \leftarrow 0]}$, $S'^1 = \{\mathsf{conf}_{[Y' \leftarrow 0]} | \mathsf{conf} \in S^1\}$ and $S'^1_\top = \{\mathsf{conf}_{[Y' \leftarrow 0]} | \mathsf{conf} \in S^1_\top\}$. Then from $v_1 \trianglerighteq_{\mathsf{set}} v_2$, we obtain $r'^1 = r'^2$, $S'^1 \supseteq S'^2$ and $S'^1_\top \subseteq S'^2_\top$ and hence $v'_1 \trianglerighteq_{\mathsf{set}} v'_2$.

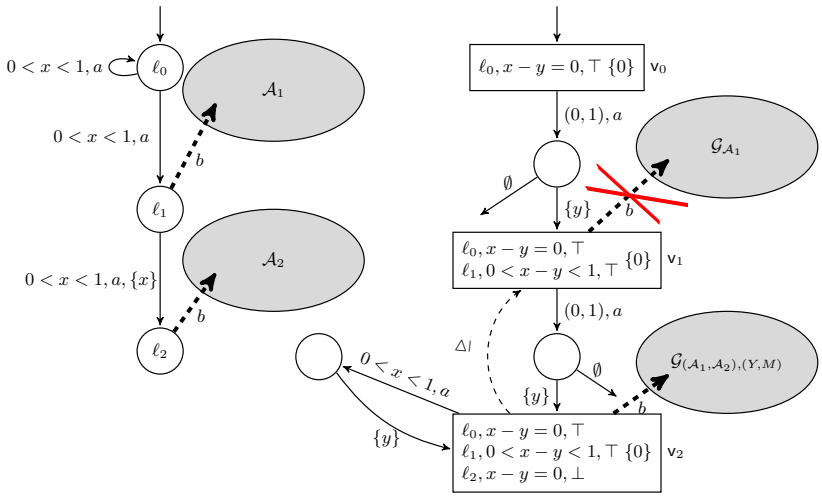

Fig. 4. Excerpt of an example of game.

– **Moves of Player** B are an action and a guard over the clocks of Y. Let (a, g) be a move of B. Then, the respective (a, g)-successors v'^1 and v'^2 of v^1 and v^2 are computed thanks to function Succ computing set of elementary successors of configurations: $v'^1 = ((S'^1, S'^1_\top), r'^1)$ and $v'^2 = ((S'^2, S'^2_\top), r'^2)$, where for $i = 1$ or 2, $r'^i = r^i$, $S'^1 = \cup_{\mathsf{conf} \in S^1} \mathsf{Succ}(\mathsf{conf}, (a, g))$, and $S'^1_\top = \cup_{\mathsf{conf} \in S^1_\top} \mathsf{Succ}_\top(\mathsf{conf}, (a, g))$, with $\mathsf{Succ}_\top$ the function Succ such that only configurations marked $\top$ are kept. From $v_1 \trianglerighteq_{\mathsf{set}} v_2$, we thus obtain that $r'^1 = r'^2$, $S'^1 \supseteq S'^2$ and $S'^1_\top \subseteq S'^2_\top$ and hence $v'_1 \trianglerighteq_{\mathsf{set}} v'_2$.

As an example, Figure 4 presents an excerpt of the construction of the game (right) for a TA (left), with $Y = \{y\}$. Gray ellipses symbolise potentially large part of the TA (resp. the game) which we do not detail here. To illustrate the tba-simulation, let us consider the two states $\mathsf{v}_2 = (\{(\ell_0, x - y = 0), (\ell_1, 0 < x - y < 1), (\ell_2, x - y = 0)\}, \{(\ell_0, x - y = 0), (\ell_1, 0 < x - y < 1)\}, \{0\})$ and $\mathsf{v}_1 = (\{(\ell_0, x - y = 0), (\ell_1, 0 < x - y < 1)\}, \{(\ell_0, x - y = 0), (\ell_1, 0 < x - y < 1)\}, \{0\})$ in the game. One can easily check that $\mathsf{v}_2 \trianglerighteq \mathsf{v}_1$. Thanks to optimisation 3 (see Section 5), Algorithm 1 applied to this example will avoid exploring $\mathcal{G}_{\mathcal{A}_1}$

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR '98*, LNCS 1466. Springer, 1998.
3. N. Bertrand, A. Stainer, T. Jéron, and M. Krichen. A game approach to determinize timed automata. In *FOSSACS'11*, LNCS 6604, Springer, 2011.

4. V. Bonifaci and A. Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. In *ESA'10*, LNCS 6347. Springer, 2010.
5. C.L. Bouton. Nim, a game with a complete mathematical theory. *Ann. Math., Ser. 2*, 3:35–39, 1902.
6. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, LNCS 3653. Springer, 2005.
7. L. Doyen and J.-F. Raskin. Antichain algorithms for finite automata. In *TACAS'10*, LNCS 6015. Springer, 2010.
8. E. Filiot, N. Jin, and JF. Raskin. Antichains and compositional algorithms for ltl synthesis. *FMSD*, 39(3):261–296, 2011.
9. G. Geeraerts, J. Goossens, and M. Lindström. Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm. *Real-Time Systems*, 49(2):171–218, 2013.
10. A. Pnueli. The temporal logic of programs. In *FOCS77*, pages 46–57. IEEE Computer Society, 1977.
11. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP89*, LNCS 372. Springer, 1989.
12. M. De Wulf, L. Doyen, N. Maquet, and J.-F. Raskin. Antichains: Alternative algorithms for ltl satisfiability and model-checking. In *TACAS'08*, LNCS 4963. Springer, 2008.

A The original OTFUR algorithm

For the sake of completeness, Algorithm 2 recalls the original OTFUR algorithm[6], adapted to the case of safety games.

Algorithm 2: OTFUR [6] algorithm for safety games

Data: G, I
`// Initialization`
Passed $:= \{I\}$; Depend$(I) := \varnothing$;
for all position v **do** Losing$[v]$:= false;
Waiting $:= \{(I, v') \in E\}$;
`// Saturation`
while Waiting $\neq \varnothing \wedge \neg$Losing$[I]$ **do**
  $e = (v, v') := pop$(Waiting);
  if $v' \notin$ Passed **then**
    Passed := Passed $\cup \{v'\}$;
    Losing$[v']$:= $v' \in$ Bad;
    Depend$[v']$:= $\{(v, v')\}$;
    if Losing$[v']$ **then**
      Waiting := Waiting $\cup \{e\}$; `// add` e `for reevaluation`
    else
      Waiting := Waiting $\cup \{(v', v'') \in E\}$;
  else
    `// reevaluation`
    Losing* := $v \in V_A \wedge \bigwedge_{v'',(v,v'')\in E}$ Losing$[v'']$
      $\vee\ v \in V_B \wedge \bigvee_{v'',(v,v'')\in E}$ Losing$[v'']$;
    if Losing* **then**
      Losing$[v]$:= true;
      Waiting := Waiting $\cup$ Depend$[v]$`// back propagation`
    if $\neg$Losing$[v']$ **then** Depend$[v']$:= Depend$[v'] \cup \{e\}$
return $\neg$Losing$[I]$