Synthesising Succinct Strategies in Safety Games with an Application to Real-time Scheduling[☆]

Gilles Geeraerts, Joël Goossens, Thi-Van-Anh Nguyen, Amélie Stainer

Département d'Informatique, Faculté des Sciences
Université libre de Bruxelles, ULB, Belgium

Abstract

We introduce general techniques to compute, *efficiently*, *succinct representations* of winning strategies in safety and reachability games. Our techniques adapt the *antichain* framework to the setting of games, and rely on the notion of *turn-based alternating simulation*, which is used to formalise natural relations that exist between the states of those games in many applications. Then, we demonstrate the applicability of our approach by considering an important problem borrowed from the *real-time scheduling* community, i.e. the problem of finding a correct schedule(r) for a set of *sporadic tasks* upon a *multiprocessor platform*. We formalise this problem by means of a game whose number of states is exponential in the description of the task set, thereby making it a perfect candidate for our approach. We have implemented our algorithm and show experimentally that it *scales* better than classical solutions. To the best of our knowledge, this is the first attempt at implementing an *exact feasibility test* for this particular problem.

Keywords: Safety games, Succinct strategies, antichains, Real-time scheduling, Sporadic tasks

1. Introduction

Finite, turn-based, games are a very simple, yet relevant, class of games. They are played by two players ($\mathcal{S}$ and $\mathcal{R}$) on a finite graph (called the arena), whose set of vertices is partitioned into Player $\mathcal{S}$ and Player $\mathcal{R}$ vertices. A play is an infinite path in this graph, 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 $\mathcal{R}$ if the token

[☆]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 n°601148 (CASSTING)

eventually touches some designated 'bad' nodes (the objective for $\mathcal{R}$ is thus a reachability objective), otherwise it is winning for $\mathcal{S}$ (for whom the objective is a safety objective), hence the names of the players.

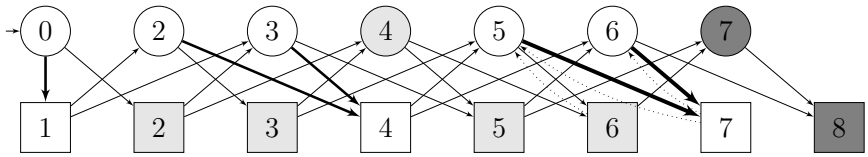
Such games are a natural model to describe the interaction of a potential controller with a given environment, where the aim of the controller (modelled by player $\mathcal{S}$) is to avoid the bad states that model system failures. They have also been used as a tool to solve other problems such as LTL realisability [1], real-time scheduler synthesis [2] or timed automata determinisation [3].

We consider, throughout the paper, a running example which is a variation of the well-known Nim game [4]. Initially, a heap of N balls is shared by the players, and the urn is empty. The players play by turn and pick either 1 or 2 balls from the heap and put them into the urn. A player loses the game if he is the last to play (i.e. the heap is empty after he has played). An arena modelling this game (for $N = 8$) is given in Figure 1 (top), where $\mathcal{S}$-states are circles, $\mathcal{R}$-states are squares, and the numbers labelling the states represent the number of balls *inside the urn*. The arena obtained from Figure 1 *without the dotted edges* faithfully models the description of the game we have sketched above (assuming Player $\mathcal{S}$ plays first). From the point of view of player $\mathcal{S}$, the set of states that he wants to avoid (and that player $\mathcal{R}$ wants to reach) is $\mathsf{Bad} = \{7, 8\}$, and we call *winning* all the states from which $\mathcal{S}$ can avoid Bad whatever $\mathcal{R}$ does. It is well-known [4] 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 $\mathcal{S}$-states v s.t. $\lambda(v) \bmod 3 \neq 1$ plus all the $\mathcal{R}$-states v' s.t. $\lambda(v') \bmod 3 = 1$.

It is well-known that *memory-less winning strategies* (i.e. that depend only on the current state) are sufficient for both players in those games. 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, the computation of winning strategies might be intractable in practice because it could request to traverse the whole arena. Moreover, a naive implementation of a winning strategy σ by means of a table mapping each $\mathcal{S}$-state v to its safe successor $\sigma(v)$ (like in Figure 1 (a) for our running example), is not realistic because this table would have the size of the arena.

In this work, we consider the problem of computing winning strategies that can be *succinctly* represented. We call '$\star$-strategy' those succinct representations, and they can be regarded as an *abstract representation* of a family of

[1] In order to make our example more interesting (this will become clear in the sequel), we have added the three *dotted edges* from 7 to 6 and 5 respectively, and from 6 to 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.



(a) Winning strategy

node	succ.	node	succ.	node	succ.	node	succ.
0	1	3	4	5	7	7	8
2	4	4	5	6	7		

(b) Winning $\star$-strategy

node	succ.	node	succ.
0	1	5	7
2	4	6	7
3	4		

(c) $\trianglerighteq_0$-Winning $\star$-strategy

node	succ.
5	7
6	7

Figure 1: Urn-filling Nim game with $N = 8$, and three winning strategies. (a) is a winning strategy mapping *all* $\mathcal{S}$ *nodes* to a suitable successor. (b) is a (slightly more compact) winning $\star$-strategy σ where the nodes n s.t. $\sigma(n) = \star$ have been omitted. This strategy remembers the move to be played from the winning and reachable $\mathcal{S}$ nodes only. (c) is a $\trianglerighteq_0$-monotonic wining $\star$-strategy that remembers the moves to be played from the *maximal* winning nodes only.

(plain) strategies, that we call *concretisations* of the $\star$-strategies. In order to keep the description of winning $\star$-strategies succinct, and to obtain efficient algorithms to compute them, we propose heuristics inspired from the *antichain* line of research [5]. These heuristics have been developed mainly in the *verification setting*, to deal with automata-based models. Roughly speaking, they rely on a *simulation partial order* on the states of the system, which is exploited to *prune* the state space that the algorithms need to explore, and to obtain *efficient data structures* to store the sets of states that the algorithms need to maintain. They have been applied to several problems, such as LTL model-checking [5] or multi-processor schedulability [6] with remarkable performance improvements of several orders of magnitude.

In this paper, we introduce general antichain-based techniques for solving safety games, and computing *efficiently succinct representations of winning strategies*. We propose a general and elegant theory which is built on top of the notion of *turn-based alternating simulation* (tba-simulation for short, a notion adapted from [7]), instead of *simulation*. 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 $\lambda(v) \bmod 3 = \lambda(v') \bmod 3$. Then, it is easy to see that the winning strategy of Figure 1 (a) exhibits a kind of *monotonicity* wrt $\trianglerighteq_0$: $5 \trianglerighteq_0 2$, and the winning strategy asks to put two balls in the urn in both cases. Hence, we can represent the winning strategy as in Figure 1 (c). Observe that not all concretisations of this strategy are winning. For instance, playing

$\boxed{3}$ from $\textcircled{2}$ is a losing move, but it is not compatible with $\trianglerighteq_0$ because $\boxed{3}$ is not $\trianglerighteq_0$-covered by $\boxed{7}$. Moreover, this succinct description of the strategy can be implemented straightforwardly: only the table in Figure 1 (c) needs to be stored in the controller, as $\trianglerighteq_0$ can be directly computed from the description of the states.

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. In Section 5, we present *an efficient on-the-fly algorithm to compute such succinct $\star$-strategies* (adapted from the classical OTFUR algorithm to solve reachability games [8]). Our algorithm generalises the algorithm of Filiot et al. [1], with several improvements: 1. it applies to a general class of games whose arena is equipped with a tba-simulation (not only those generated from an instance of the LTL realisability problem); 2. it contains an additional heuristic that was not present in [1]; 3. its proof of correctness is straightforward, and stems directly from the definition of tba-simulation.

Then, in Section 6 we present extensively an application of our theory to a particular problem, borrowed from the real-time scheduling community, namely the *feasibility problem for sporadic tasks on multiprocessor platforms* [9]. In this problem, a set of real-time tasks release, periodically, *jobs* that must be scheduled on a set of m CPUs by a *scheduler*. Each job is characterised by an execution time C, which is the amount of CPU time it needs to complete; and by a deadline D before which it must have been granted these C CPU time units. The duty of the scheduler is to attribute the jobs that are ready for execution to the CPUs, in order to avoid missing any deadline (of course, in general, there can be more jobs than CPUs). It can easily be modelled as a safety game, where the $\mathcal{S}$ player is the scheduler; the $\mathcal{R}$ player is the coalition of the task; and the set of bad states that the scheduler must avoid is the set of states where at least one job has missed its deadline. Then, a winning strategy is the actual scheduling policy that must be implemented to avoid deadline misses. So we formalise this problem by means of *scheduling games*. Then, building on our previous works on the related feasibility problem for sporadic tasks [10], we propose a tba-simulation for scheduling games. Finally, in Section 7, we demonstrate, by means of experiments that our optimised version of the OTFUR algorithm scales much better in practice that the exhaustive search, and that the basic version of OTFUR (i.e. without the optimisation based on tba-simulation). To the best of our knowledge, this is the first attempt at implementing an exact feasibility test for sporadic tasks on a multiprocessor platform.

2. Preliminaries

Turn-based games. A *finite turn-based game arena* is a tuple $\mathcal{A} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I)$, where $V_{\mathcal{S}}$ and $V_{\mathcal{R}}$ are the finite sets of states controlled by Players $\mathcal{S}$ and $\mathcal{R}$ respectively; $E \subseteq (V_{\mathcal{S}} \times V_{\mathcal{R}}) \cup (V_{\mathcal{R}} \times V_{\mathcal{S}})$ is the set of edges; and $I \in V_{\mathcal{S}}$ is the initial state. We let $V = V_{\mathcal{S}} \cup V_{\mathcal{R}}$. For a finite arena $\mathcal{A} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I)$ 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^* is the reflexive and transitive closure of E. We write

$\mathsf{Reach}(\mathcal{A})$ instead of $\mathsf{Reach}(\mathcal{A}, I)$, and lift the definitions of Reach and Succ to sets of states in the usual way.

The aim of Player $\mathcal{R}$ is to *reach* some designated set of states Bad, while the aim of $\mathcal{S}$ is to *avoid* it. Throughout this paper, we focus on the objective of player $\mathcal{S}$, and regard our finite games as *safety games* because they correspond to the applications we target (see Section 6 and the discussion of other potential application in the conclusion). However, safety and reachability games are symmetrical and determined, so, our results can easily be adapted to cope with *reachability games*. Formally, A *finite turn-based (safety) game* is a tuple $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ where $(V_{\mathcal{S}}, V_{\mathcal{R}}, E, I)$ is a finite turn-based game arena, and $\mathsf{Bad} \subseteq V$ is the set of bad states that $\mathcal{S}$ wants to avoid. The definitions of Reach and Succ carry over to games: for a game $\mathcal{G} = (\mathcal{A}, \mathsf{Bad})$, we let $\mathsf{Reach}(\mathcal{G}, v) = \mathsf{Reach}(\mathcal{A}, v)$, $\mathsf{Reach}(\mathcal{G}) = \mathsf{Reach}(\mathcal{A})$ and $\mathsf{Succ}(\mathcal{G}, v) = \mathsf{Succ}(\mathcal{A}, v)$. 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 $\mathcal{S}$ is a function $\sigma : V_{\mathcal{S}} \to V_{\mathcal{R}}$ such that for all $v \in V_{\mathcal{S}}$, $(v, \sigma(v)) \in E$. We extend strategies to set of states S in the usual way: $\sigma(S) = \{\sigma(v) \mid v \in S\}$. A strategy σ for $\mathcal{S}$ is *winning for a state* $v \in V$ iff no bad states are reachable from v in the graph $\mathcal{G}_\sigma$ obtained from $\mathcal{G}$ by removing all the moves of $\mathcal{S}$ which are not chosen by σ, i.e. $\mathsf{Reach}(\mathcal{G}_\sigma, v) \cap \mathsf{Bad} = \emptyset$, where $\mathcal{G}_\sigma = (V_{\mathcal{S}}, V_{\mathcal{R}}, E_\sigma, I, \mathsf{Bad})$ and $E_\sigma = \{(v, v') \mid (v, v') \in E \wedge v \in V_{\mathcal{S}} \implies v' = \sigma(v)\}$. We say that a strategy σ is *winning* in a game $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ iff it is winning in $\mathcal{G}$ for I.

Winning states and attractor. A state $v \in V$ in $\mathcal{G}$ is *winning* (for Player $\mathcal{S}$) iff there exists a strategy σ that is winning in $\mathcal{G}$ for v. We denote by Win the set of winning states (for Player $\mathcal{S}$). By definition, any strategy such that $\sigma(\mathsf{Win}) \subseteq \mathsf{Win}$ is thus winning. Moreover, it is well-known that Win can be computed in polynomial time (in the size of the arena), by computing the so-called *attractor* (for Player $\mathcal{R}$) of the unsafe states. In a game $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$, 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 $i \in \mathbb{N}$, $\mathsf{Attr}_{i+1} = \mathsf{Attr}_i \cup \{v \in V_{\mathcal{R}} \mid \mathsf{Succ}(v) \cap \mathsf{Attr}_i \neq \emptyset\} \cup \{v \in V_{\mathcal{S}} \mid \mathsf{Succ}(v) \subseteq \mathsf{Attr}_i\}$. 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 $\mathcal{R}$ can force the game to reach Bad from v. Thus, the set of winning states for Player $\mathcal{S}$ is $\mathsf{Win} = V \setminus \mathsf{Attr}(\mathsf{Bad})$. Then, the strategy σ s.t. for all $v \in V_{\mathcal{S}} \cap \mathsf{Win}$, $\sigma(v) = v'$ with $v' \in \mathsf{Win}$ is winning.

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 $s, s' \in \alpha$: $s \neq s'$ implies $s \not\trianglerighteq s'$. 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 $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$. A relation $\trianglerighteq \subseteq V_{\mathcal{S}} \times V_{\mathcal{S}} \cup V_{\mathcal{R}} \times V_{\mathcal{R}}$ 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 $\trianglerighteq_0 = \{(v, v') \in V_{\mathcal{S}} \times V_{\mathcal{S}} \cup V_{\mathcal{R}} \times V_{\mathcal{R}} \mid \lambda(v) \geq \lambda(v') \text{ and } \lambda(v) \bmod 3 = \lambda(v') \bmod 3\}$ is a simulation relation compatible with $\mathsf{Bad} = \{\textcircled{7}, \boxed{8}\}$. Moreover, $\mathsf{Win} = \{v \in V_{\mathcal{S}} \mid \lambda(v) \bmod 3 \neq 1\} \cup \{v \in V_{\mathcal{R}} \mid \lambda(v) \bmod 3 = 1\}$ 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 our notion of *succinct strategy* (observe that other works propose different notions of 'small strategies', see for instance [11]). 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 $\{(v, \sigma(v)) \mid v \in V_{\mathcal{S}}\}$, and implement a controller that traverses the whole table to find action to perform

[2] See [1] 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.

each time the system state is updated. While the definition of strategy asks that $\sigma(v)$ be defined for all $\mathcal{S}$-states v, this information is sometimes redundant, for instance, when v is not reachable in $\mathcal{G}_\sigma$. 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 $\hat{\sigma} : V_{\mathcal{S}} \mapsto V_{\mathcal{R}} \cup \{\star\}$, where $\star$ stands for a 'don't care' information. We denote by $\mathsf{Supp}(\hat{\sigma})$ the *support* $\hat{\sigma}^{-1}(V_{\mathcal{R}})$ 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* of a $\star$-strategy $\hat{\sigma}$ is a strategy σ s.t. for all $v \in V_{\mathcal{S}}$, $\hat{\sigma}(v) \neq \star$ 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 $\mathcal{S}$ 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 Figure 1 (b)—assuming the lines where $\sigma(v) = \star$ have been omitted—is a winning $\star$-strategy of minimal size.

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 game $\mathcal{G}$ and an integer $k \in \mathbb{N}$, decide whether there is a winning $\star$-strategy of size smaller than k in $\mathcal{G}$.

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 propositional 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 $\mathcal{G}$ and an integer k in polynomial time as follows. The construction of $\mathcal{G}$ is illustrated in Figure 2 for $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ (hence $k = 8$). Note that the state 'bad' and the $(\neg)x^{\mathcal{R}}_i$'s have been duplicated to enhance readability

Formally, we let $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ where:

- $V_{\mathcal{S}} = \{\text{init}^{\mathcal{S}}\} \cup \mathcal{L}^{\mathcal{S}} \cup X \cup C$, where $\mathcal{L}^{\mathcal{S}} = \{x^{\mathcal{S}}_1, \cdots, x^{\mathcal{S}}_m, \neg x^{\mathcal{S}}_1, \cdots, \neg x^{\mathcal{S}}_m\}$, $\mathcal{X} = \{X_1, \cdots, X_m\}$ and $\mathcal{C} = \{C_1, \cdots, C_n\}$
- $V_{\mathcal{R}} = \{\text{init}^{\mathcal{R}}, \text{bad}\} \cup \{x^{\mathcal{R}}_1, \cdots, x^{\mathcal{R}}_m, \neg x^{\mathcal{R}}_1, \cdots, \neg x^{\mathcal{R}}_m\}$
- $E = \{(\text{init}^{\mathcal{S}}, \text{init}^{\mathcal{R}})\} \cup \{\text{init}^{\mathcal{R}}\} \times (\mathcal{X} \cup \mathcal{C})$
  $\cup \left\{ \begin{array}{c} (X_i, x^{\mathcal{R}}_i), (X_i, \neg x^{\mathcal{R}}_i), (x^{\mathcal{R}}_i, x^{\mathcal{S}}_i), (\neg x^{\mathcal{R}}_i, \neg x^{\mathcal{S}}_i), \\ (x^{\mathcal{S}}_i, \text{init}^{\mathcal{R}}), (\neg x^{\mathcal{S}}_i, \text{init}^{\mathcal{R}}), (x^{\mathcal{S}}_i, \text{bad}), \\ (\neg x^{\mathcal{S}}_i, \text{bad}), (X_i, \text{bad}), (C_i, \text{bad}) \end{array} \middle| \; 1 \leq i \leq m \right\}$
  $\cup (\mathcal{L}^{\mathcal{S}} \cup \mathcal{C} \cup \mathcal{X}) \times \{\text{bad}\}$
  $\cup \{(C_i, x^{\mathcal{R}}_h) \mid 1 \leq i \leq n, \exists 1 \leq j \leq n_i : l^i_j = x_h\}$
  $\cup \{(C_i, \neg x^{\mathcal{R}}_h) \mid 1 \leq i \leq n, \exists 1 \leq j \leq n_i : l^i_j = \neg x_h\}$

- $I = \text{init}^{\mathcal{S}}$
- $\mathsf{Bad} = \{\text{bad}\}$

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 $(\neg)x_i^{\mathcal{R}}$'s) have been duplicated to enhance readability). States of $\mathcal{S}$ are circles and states of $\mathcal{R}$ are squares.

Let us show that ϕ is satisfiable iff there is a winning $\star$-strategy of size at most k in $\mathcal{G}$. To this end, we first make several observations on $\mathcal{G}$. First, there is a winning strategy in $\mathcal{G}$ since all predecessors of *bad* are Player $\mathcal{S}$ states that have other successors that are not *bad*. Second, Player $\mathcal{S}$ can never avoid the states of the form X_i nor C_j, i.e. for all strategy σ: $\mathcal{X} \cup \mathcal{C} \subseteq \mathsf{Reach}\left(\mathcal{G}_\sigma, \text{init}^{\mathcal{S}}\right)$. 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 $\mathcal{G}_\sigma$ by the path $\text{init}^{\mathcal{S}}, \text{init}^{\mathcal{R}}, v, \text{bad}$, 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 $x_i^{\mathcal{R}}$, or to $\neg x_i^{\mathcal{R}}$, and those nodes cannot be avoided by Player $\mathcal{S}$, 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 $\mathcal{L}^{\mathcal{S}}$. More precisely, for all winning $\star$-strategies $\hat{\sigma}$, let $S_{\hat{\sigma}} = \{x_i^{\mathcal{S}} \mid \exists 1 \leq i \leq m : \hat{\sigma}(X_i) = x_i^{\mathcal{R}}\} \cup \{\neg x_i^{\mathcal{S}} \mid \exists 1 \leq i \leq m : \hat{\sigma}(X_i) = \neg x_i^{\mathcal{R}}\}$. 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 $\hat{\sigma}$ has size at least $k = 2 \times m + n$ in $\mathcal{G}$.

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 $1 \leq i \leq m$: $\hat{\sigma}(X_i) = x_i^{\mathcal{R}}$ if $a(x_i) = \mathbf{true}$ and $\hat{\sigma}(X_i) = \neg x_i^{\mathcal{R}}$ 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 $\hat{\sigma}(C_i) = \ell_i^{\mathcal{R}}$. For all $v \in \{x_1^{\mathcal{S}}, \neg x_1^{\mathcal{S}}, \ldots, x_n^{\mathcal{S}}, \neg x_n^{\mathcal{S}}\}$, let $\hat{\sigma}(v) = \text{init}^{\mathcal{R}}$ if $v = \hat{\sigma}(X_i)$ for some $1 \leq i \leq n$, and let $\hat{\sigma}(v) = \star$ otherwise. Finally, let $\hat{\sigma}(\text{init}^{\mathcal{S}}) = \star$. 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 $\overline{\sigma}(X_i) = x_i^{\mathcal{R}}$. 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 $\mathcal{G}$, 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 $\overline{\sigma}(v) = \sim x_k^{\mathcal{R}}$ for some k, and where $\sim$ can be $\neg$ or nothing. Then, by definition of $\mathcal{G}$, the corresponding $\mathcal{S}$ state $\sim x_k^{\mathcal{S}}$ is reachable in $\mathcal{G}_\sigma$ for all concretisations σ of $\overline{\sigma}$. Moreover, $\sim x_k^{\mathcal{S}} \notin X \cup C \cup S_{\overline{\sigma}}$, hence $\overline{\sigma}(\sim x_k^{\mathcal{S}}) = \star$, and there is at least one concretisation of $\overline{\sigma}$ that maps $\sim x_k^{\mathcal{S}}$ to bad. Since $\sim x_k^{\mathcal{S}}$ is reachable in this

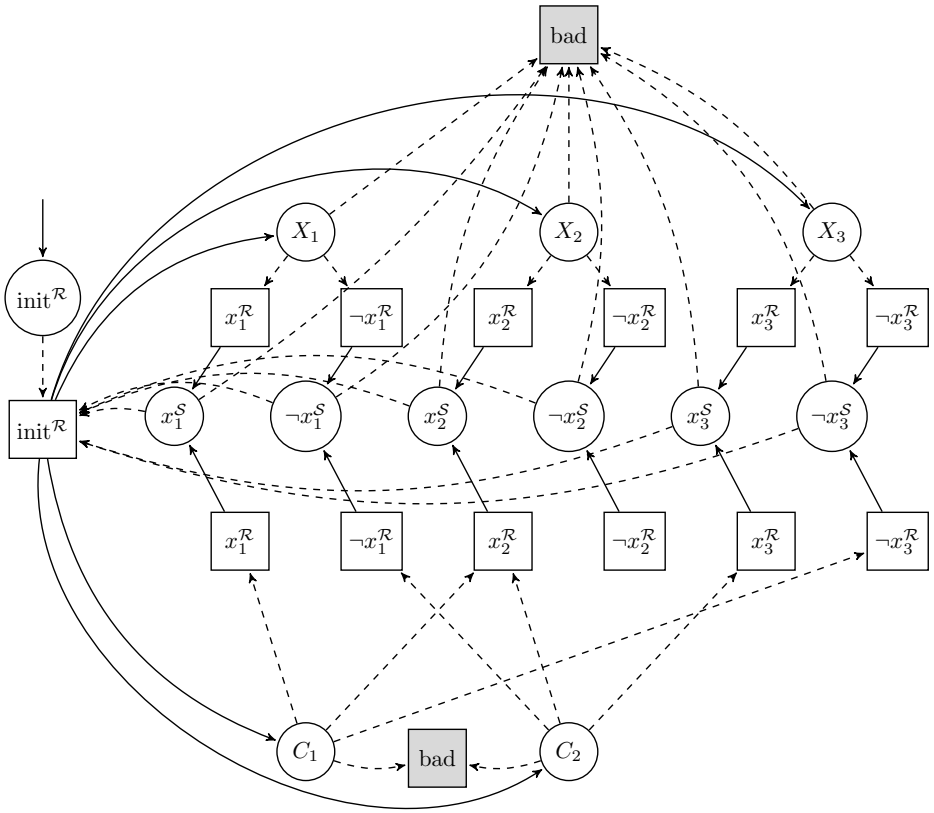


Figure 2: Construction for the formula $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$: $k = 8$. State *bad*, and all the states of the form $(\neg)x_i^{\mathcal{R}}$ have been duplicated for readability.

concretisation in particular, we conclude that $\overline{\sigma}$ is not winning. Contradiction. □

Note that this hardness result is even exacerbated in most practical cases we are aware of, since the arena is usually not given explicitly. This is the case in particular with the real-time sporadic task feasibility problem we consider in Section 6 and Section 7, and with the other potential applications that we discuss in the conclusion, because they can be reduced to safety games whose sizes are 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, 5, 1]. In this line of research, the authors point out that, in practical applications (like the scheduling problem of 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 finite turn-based games and (ii) obtain succinct representations of strategies. This notion is adapted from [7]. Here is its formal definition:

Definition 1. *Let* $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ *be a finite safety game. A partial order* $\trianglerighteq \subseteq V_{\mathcal{S}} \times V_{\mathcal{S}} \cup V_{\mathcal{R}} \times V_{\mathcal{R}}$ *is a* turn-based alternating simulation relation for $\mathcal{G}$ *[7] (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* $v_1 \in V_{\mathcal{S}}$, *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* $v_1 \in V_{\mathcal{R}}$, *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 (Figure 1), $\trianglerighteq_0$ is a tba-simulation relation. Indeed, as we are going to see at then end of the present section, a simulation relation in a game where player $\mathcal{S}$ 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 $v \in V_{\mathcal{S}}$: (i) $v \in \mathsf{Supp}(\hat{\sigma})$ **implies** $\sigma(v) = \hat{\sigma}(v)$; and (ii) $\big(v \notin \mathsf{Supp}(\hat{\sigma}) \wedge v \in {\downarrow^{\trianglerighteq}}(\mathsf{Supp}(\hat{\sigma}))\big)$ **implies** $\exists \overline{v} \in \mathsf{Supp}(\hat{\sigma})$ s.t. $\overline{v} \trianglerighteq v$ and $\sigma(\overline{v}) \trianglerighteq \sigma(v)$. 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 Figure 1 (b) and highlighted by bold arrows in Figure 1 (thus, $\overline{\sigma}(\textcircled{4}) = \overline{\sigma}(\textcircled{7}) = \star$). Yet, one can define a $\trianglerighteq_0$-winning $\star$-strategy $\hat{\sigma}$ of size 2 because states $\textcircled{5}$ and $\textcircled{6}$ simulate all the winning states of $\mathcal{S}$. This $\star$-strategy[4] $\hat{\sigma}$ is the one given in Figure 1 (c) and represented by the boldest arrows in Figure 1. Observe that, while all $\trianglerighteq$-concretisations of $\hat{\sigma}$ are winning, not all *concretisations* of $\hat{\sigma}$ are. For instance, all concretisations σ of $\hat{\sigma}$ s.t. $\sigma(\textcircled{0}) = \boxed{2}$ are not $\trianglerighteq_0$-monotonic and losing.

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

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 a maximal antichain of the winning states. In Section 5, 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 below. For all strategies σ, and all $V \subseteq V_{\mathcal{S}}$, 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 $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ be a finite turn-based game and $\trianglerighteq$ be a tba-simulation relation for $\mathcal{G}$. Let σ be a strategy in $\mathcal{G}$, and let $\mathbf{S} \subseteq V_{\mathcal{S}}$ be a set of $\mathcal{S}$-states s.t.: (i) $(\mathbf{S} \cup \sigma(\mathbf{S})) \cap \mathsf{Bad} = \emptyset$; (ii) $I \in {\downarrow^{\trianglerighteq}}(\mathbf{S})$; and (iii) $\mathsf{succ}(\sigma(\mathbf{S})) \subseteq {\downarrow^{\trianglerighteq}}(\mathbf{S})$. Then, $\sigma|_{\mathbf{S}}$ is a $\trianglerighteq$-winning $\star$-strategy.*

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

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

We conclude the proof by observing that $\tau(\mathbf{S}) = \sigma(\mathbf{S})$, since τ is a $\trianglerighteq$-concretisation of $\sigma|_{\mathbf{S}}$. Hence, ${\downarrow}(\mathbf{S} \cup \tau(\mathbf{S})) = {\downarrow}(\mathbf{S} \cup \sigma(\mathbf{S}))$. Moreover, hypothesis (i) implies that ${\downarrow}(\mathbf{S} \cup \sigma(\mathbf{S})) \cap \mathsf{Bad} = \emptyset$, by definition of tba-simulation (compatible with Bad). Hence, since $\mathsf{Reach}(\mathcal{G}_\tau) \subseteq {\downarrow}(\mathbf{S} \cup \tau(\mathbf{S})) = {\downarrow}(\mathbf{S} \cup \sigma(\mathbf{S}))$ by the arguments above, we conclude that $\mathsf{Reach}(\mathcal{G}_\tau) \cap \mathsf{Bad} = \emptyset$, 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 $\mathcal{S}$-states, for a given winning strategy σ:

Theorem 2. *Let $\mathcal{G} = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ be a finite turn-based game, $\trianglerighteq$ be a tba-simulation relation for $\mathcal{G}$. Let σ be a winning strategy and $\mathcal{WR}_\sigma$ be a maximal $\trianglerighteq$-antichain on $\mathsf{Reach}(\mathcal{G}_\sigma) \cap V_\mathcal{S}$, 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) $\mathcal{WR}_\sigma \cup \sigma(\mathcal{WR}_\sigma) \subseteq \mathsf{Reach}(\mathcal{G}_\sigma)$ by definition of $\mathcal{WR}_\sigma$ and $\mathsf{Reach}\,(\mathcal{G}_\sigma) \cap \mathsf{Bad} = \emptyset$ because σ is winning. Hence $(\mathcal{WR}_\sigma \cup \sigma(\mathcal{WR}_\sigma)) \cap \mathsf{Bad} = \emptyset$. (ii) $I \in \mathsf{Reach}\,(\mathcal{G}_\sigma) \cap V_\mathcal{S}$ and $\mathsf{Reach}\,(\mathcal{G}_\sigma) \cap V_\mathcal{S} \subseteq{\downarrow}\,(\mathcal{WR}_\sigma)$ by definition, hence $I \in{\downarrow}\,(\mathcal{WR}_\sigma)$. (iii) $\mathsf{Succ}\,(\sigma)\,(\mathcal{WR}_\sigma)) \subseteq \mathsf{Reach}\,(\mathcal{G}_\sigma)$ and $\mathsf{Reach}\,(\mathcal{G}_\sigma) \subseteq{\downarrow}\,(\mathcal{WR}_\sigma)$ by definition, hence $\mathsf{Succ}\,(\sigma(\mathcal{WR}_\sigma)) \subseteq{\downarrow}\,(\mathcal{WR}_\sigma)$. □

5. Efficient computation of succinct winning strategies

Let us now present an efficient algorithm to compute succinct strategies in safety games. Our algorithm is an optimisation of the OTFUR algorithm [8], which is an on-the-fly algorithm for reachability (hence also for safety) games. The on-the-fly features of OTFUR permits, in the best cases, to avoid exploring (and building) some parts of the game graph, and returns a winning strategy as soon as one is found. We introduce more aggressive optimisations exploiting the properties of tba-simulations. Our algorithm generalises the algorithm of Filiot *et al.* [1], and our proofs, that rely on the definition of tba-simulations, are also more elegant and straightforward.

5.1. The original OTFUR algorithm

The *On-The-Fly algorithm for Untimed Reachability games* (OTFUR) algorithm [8] is an efficient, on-the-fly algorithm to compute a winning strategy *for Player* $\mathcal{R}$, i.e. when considering a *reachability objective*. It is easy to adapt it to compute winning strategies for Player $\mathcal{S}$ instead. We sketch the main ideas behind this algorithm, and refer the reader to [8] for a comprehensive description. 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 optimised version of OTFUR for games equipped with tba-simulations. Before this, we prove that, when a finite turn-based 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.

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

Data: A finite turn-based game $\mathcal{G} = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$

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

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

To prove this proposition, we start by an auxiliary lemma that relates tba-simulations and computation of the attractor set:

Lemma 1. *Let $\mathcal{G}$ be a finite safety game, and let $\trianglerighteq$ be an tba-simulation for $\mathcal{G}$. 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:

- In the case where $v_2 \in V_{\mathcal{R}}$: first of all $v_1 \trianglerighteq v_2$ implies $v_1 \in V_{\mathcal{R}}$. Moreover by definition of Attr_i, there exists $v_2' \in V_{\mathcal{S}}$ 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 $v_1' \in V_{\mathcal{S}}$ 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 $v_2 \in V_{\mathcal{S}}$: first of all $v_1 \trianglerighteq v_2$ implies $v_1 \in V_{\mathcal{S}}$. Moreover by definition of Attr_i, for all $v_2' \in V_{\mathcal{R}}$ 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 $v_1' \in V_{\mathcal{R}}$ such that $(v_1, v_1') \in E$, then there exists $v_2'' \in V_{\mathcal{R}}$ 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 the induction hypothesis $v_1' \in \mathsf{Attr}_{i-1}$. Hence by definition of Attr_i, $v_1 \in \mathsf{Attr}_i$. □

Now, we can prove Proposition 2:

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 $\trianglerighteq$-downward closed. □

5.2. Optimised version of OTFUR

Let us discuss Algorithm 1, our optimised version of OTFUR for the construction of $\trianglerighteq$-winning $\star$-strategies. Its high-level principle is the same as 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[5]. 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 exploring this edge can

[5]We could initialise AntiLosingto Bad, but this is not always practical. In particular, when the arena is not given explicitly, we want to avoid pre-computing Bad.

be postponed (lines 8–18, see below). Then, if v' has not been explored before (line 19), cannot be declared surely losing (line 21), and does not belong to Bad (line 22), then it is explored (lines 25–31). If v' is found to be losing, e is put back in Waiting for back propagation (either at 24 or at line 33). The actual back-propagation is performed in lines 34–41 and triggered by an edge (v, v') s.t. $v' \in$ Passed. Let us highlight the three optimisations that rely on a tba-simulation $\trianglerighteq$:

1. By the properties of $\trianglerighteq$, we explore only the $\trianglerighteq$-minimal (respectively $\trianglerighteq$-maximal) successors of each $\mathcal{S}$ ($\mathcal{R}$) state (see lines 4, 29 and 31). We consider maximal and minimal elements only when evaluating a node in line 35.

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 surely losing as an antichain AntiLosing of minimal losing states.

3. Symmetrically, the set of *possibly winning states* is stored as an antichain 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 Waiting. Before exploring it, we first check whether either v or v' belongs to $\downarrow$(AntiMaybe) (see lines 8 and 12). If yes, there is $v_m \in$ 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 Depend$[v_m]$, so that, if v_m is eventually declared losing (see line 39), (v, v') will be re-scheduled for exploration. Thus, the algorithm stops when all maximal $\mathcal{S}$ states have a successor that is covered by a non-losing one.

Observe that optimisations 1 and 2 rely on the upward closure of the losing states only, and were present in the antichain algorithm of [1]. 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, $I \not\in\!\uparrow$(AntiLosing), we can extract from AntiMaybe a winning $\star$-strategy $\hat{\sigma}_{\mathcal{G}}$ as follows. For all $v \in$ AntiMaybe$\cap V_{\mathcal{S}}$, we let $\hat{\sigma}_{\mathcal{G}}(v) = v'$ such that $v' \in$ Succ$(v) \cap \downarrow$(AntiMaybe). For all $v \in V_{\mathcal{S}} \setminus$ AntiMaybe, we let $\hat{\sigma}_{\mathcal{G}}(v) = \star$. Symmetrically, if $I \in\!\uparrow$(AntiLosing), there is no winning strategy for $\mathcal{S}$.

Theorem 3. *When called on game $\mathcal{G}$, Algorithm 1 always terminates. Upon termination, either $I \in\!\uparrow$(AntiLosing) and there is no winning strategy for $\mathcal{S}$ in $\mathcal{G}$, or $\hat{\sigma}_{\mathcal{G}}$ 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** loop in Algorithm 1;

so, for instance, $\mathsf{Waiting}_i$ denotes the set Waiting at the i-th iteration. Let us define two further notations, for all i: $\mathsf{StateWaiting}_i = \{v \mid \exists v', (v, v') \in \mathsf{Waiting}_i \text{ or } (v', v) \in \mathsf{Waiting}_i\}$, $\mathsf{Visited}_i = \{v \mid \exists j \leq i, v \in \mathsf{StateWaiting}_j\}$, and $S_i = \mathsf{AntiMaybe}_i \cup \mathsf{StateWaiting}_i$. In words, $\mathsf{StateWaiting}_i$ is the set of states which appear in Waiting at the i-th iteration of **while**, $\mathsf{Visited}_i$ is the set of the states which have appeared in 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 AntiMaybe, Visited, StateWaiting, AntiLosing and 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 indexes $i < j$, $\mathsf{Passed}_i \subseteq \mathsf{Passed}_j$ 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. $\mathsf{Passed}_i = \mathsf{Passed}_{i-1}$ and $\uparrow(\mathsf{AntiLosing}_i) = \uparrow\big(\mathsf{AntiLosing}_{i-1}\big)$) then Waiting decreases strictly, i.e. $\mathsf{Waiting}_i \subseteq \mathsf{Waiting}_{i-1}$. Indeed, elements can be added to Waiting only in lines 24, 29, 31, 39, or 33. In the last case (Waiting is modified in line 33), AntiLosing has necessarily increased in line 37. Indeed, in this line, we are sure that v is not already in $\uparrow(\mathsf{Antilosing})$, because of the test in line 7. In the other cases, Passed has necessarily increased in line 20. Thus, each addition to Waiting necessarily implies an addition to either Passed or AntiLosing, which implies that if Passed and AntiLosing stay constant, then no element is added to Waiting. However, since at least one element is removed from waiting at each iteration (line 6), we conclude that, *if* Passed and AntiLosing stay constant, *then* Passed decreases. 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. To prove that the algorithm indeed computes a winning strategy (when one exists) we first establish five

loop invariants, that hold for all i:

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

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

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

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

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

Before proving these invariants, let us give some intuitions about them. Inv_i^1 says that all visited states (that are not pending exploration in Waiting) are categorised either as surely losing (in $\uparrow(\mathsf{AntiLosing}_i)$), or as potentially winning (in $\downarrow(\mathsf{AntiMaybe}_i)$). Inv_i^2 and Inv_i^3 tell us that, although we have not fully developed all the successors of all nodes, we are nonetheless keeping track of enough information. More precisely, Inv_i^2 says that, for all Player $\mathcal{S}$ node v which is a maximal, potentially winning node ($v \in \mathsf{AntiMaybe}$), there is always at least one successor v' s.t. either v' is itself regarded as potentially winning ($v' \in \downarrow(\mathsf{AntiMaybe})$), or the edge (v, v') is in the dependency list of some potentially winning node ($(v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe})]$), or the edge (v, v') has simply not been inspected yet and is thus in Waiting. Inv_i^3 is symmetrical for a Player $\mathcal{R}$ node v: in this case, all successors of v should satisfy one of the three previous conditions (otherwise, it would imply that v has a successor which has been detected as surely losing, hence v should be declared surely losing too and move to AntiLosing). Inv_i^4 tell us that all nodes that we insert in AntiLosing are surely losing (remember that nodes in AntiMaybe are not surely winning as already discussed). Finally, Inv_i^5 tells us that, whenever an edge (v, v') is in the dependency list of some node $\overline{v}$, then, either $\overline{v}$ covers v', or it strictly covers v.

Let us now prove these invariants.

Proof (of invariant Inv_i^1). We start by observing that the sets $\uparrow(\mathsf{AntiLosing})$ and Visited increase monotonically along the execution of the algorithm.

At $i = 0$, all the states in Visited are in StateWaiting, hence the initialisation 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 $\big(\mathsf{Visited}_{i+1} \setminus \mathsf{StateWaiting}_{i+1}\big) \cap \big(\mathsf{Visited}_i \setminus \mathsf{StateWaiting}_i\big)$. 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 $v \in \uparrow\big(\mathsf{AntiLosing}_{i+1}\big)$ (see the beginning 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 that: $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow(\mathsf{AntiLosing})_{i+1}$, 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 38, 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 39), 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 6, 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 31 and 29 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 20. One can also check that, in this case, 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, $v = v_2$, and either $v \in \downarrow(\mathsf{AntiMaybe})_{i+1} \cup \uparrow\big(\mathsf{AntiLosing}_{i+1}\big)$, or v is not passed yet and the conditional in line 19 is satisfied. As a consequence, at the end of the iteration, $v \in \mathsf{AntiMaybe}_{i+1}$ or $v \in \mathsf{AntiLosing}_{i+1}$. □

Proof (of invariant Inv_i^2). At $i = 0$, the only state in $\mathsf{AntiMaybe}$ is I and $\mathsf{Waiting}$ contains all the edges of the form (I, v) where v is minimal wrt $\trianglerighteq$. Hence, there is at least one successor v of I s.t. $(I, v) \in \mathsf{Waiting}$ and the invariant holds for $i = 0$.

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. Let v be a node in $\mathsf{AntiMaybe}_{i+1} \cap V_{\mathcal{S}}$ and let us show that there exists $v' \in \mathsf{Succ}(v)$ s.t. either $v' \in \downarrow\big(\mathsf{AntiMaybe}_{i+1}\big)$ or $(v, v') \in \mathsf{Depend}_{i+1}[\downarrow\big(\mathsf{AntiMaybe}_{i+1}\big)] \cup \mathsf{Waiting}_{i+1}$. We consider several cases. For the sake of clarity, we use the ✓symbol to mean that a case is closed. First, we assume that $v \in \downarrow\big(\mathsf{AntiMaybe}_{i+1}\big) \setminus \downarrow(\mathsf{AntiMaybe}_i)$. In this case, line 29 has been executed (since $v \in V_{\mathcal{S}}$), and for all $v' \in \lfloor\mathsf{Succ}(v)\rfloor$: $(v, v') \in \mathsf{Waiting}_{i+1}$, hence the

invariant holds (✓). Thus, it remains to consider the cases where v was already in $\mathsf{AntiMaybe}_i$. In this case, we know, by induction hypothesis, that there exists $v' \in \mathsf{Succ}(v)$ s.t. either $v' \in \downarrow(\mathsf{AntiMaybe}_i)$ or $(v, v') \in \mathsf{Depend}_i[\downarrow(\mathsf{AntiMaybe}_{i+1})]$ or $(v, v') \in \mathsf{Waiting}_i$. Let us consider these three cases separately:

1. Clearly, if $v' \in \downarrow(\mathsf{AntiMaybe}_{i+1})$, then the invariant holds. (✓). Thus, we assume that $v' \in \downarrow(\mathsf{AntiMaybe}_i) \setminus \downarrow(\mathsf{AntiMaybe}_{i+1})$. It is easy to check that, unless line 38 has been executed, $\downarrow(\mathsf{AntiMaybe})$ grows along the iterations. Hence, the fact that $v' \in \downarrow(\mathsf{AntiMaybe}_i) \setminus \downarrow(\mathsf{AntiMaybe}_{i+1})$ implies that line 38 has been executed during the i-th iteration of the loop. Let us better characterise what occurs when line 38 is executed. First, observe that the set $\uparrow(\mathsf{AntiLosing})$ grows strictly, because a node $\overline{v}$ (called v in the code of the algorithm) is added $\mathsf{AntiLosing}$ in line 37. Moreover, at all times, all nodes in Passed are either in $\uparrow(\mathsf{AntiLosing})$ or in $\downarrow(\mathsf{AntiMaybe})$ (by Inv_i^1). Finally, by inspecting the different conditionals that must hold for line 38 to be executed, one can check that the node $\overline{v}$ is necessarily a maximal node of $\downarrow(\mathsf{AntiMaybe})$ before executing line 38, i.e. $\overline{v} \in \mathsf{AntiMaybe}_i \setminus \mathsf{AntiMaybe}_{i+1}$. From these facts, let us show that $\downarrow(\mathsf{AntiMaybe}_{i+1}) = \downarrow(\mathsf{AntiMaybe}_i) \setminus \{\overline{v}\}$, i.e. $\overline{v}$ is the only node which is removed from $\downarrow(\mathsf{AntiMaybe})$ when executing line 38. Indeed, after the execution of line 38, we have:

$$\begin{aligned}\downarrow(\mathsf{AntiMaybe}_{i+1}) &= \downarrow(\mathsf{Passed}_i \setminus \uparrow(\mathsf{AntiLosing}_{i+1})) \\ &= \downarrow(\mathsf{Passed}_i \setminus (\uparrow(\mathsf{AntiLosing}_i \cup \{\overline{v}\})))\end{aligned}$$

Now let us consider a state $w \in \mathsf{Passed}_i$. There are two possibilities: either $w \in \downarrow(\mathsf{AntiMaybe}_i)$, or $w \in \uparrow(\mathsf{AntiLosing}_i)$. In the first case: $w \in \downarrow(\mathsf{AntiMaybe}_i)$, then $w \notin \uparrow(\mathsf{AntiLosing}_i)$. Hence, $w \in \downarrow(\mathsf{AntiMaybe}_i \setminus \{\overline{v}\})$. However, since $\overline{v} \in \mathsf{AntiMaybe}_i$, this is equivalent to $w \in \downarrow(\mathsf{AntiMaybe}_i) \setminus \{\overline{v}\}$. In the latter case, $w \in \uparrow(\mathsf{AntiLosing}_i)$, it is clear $w \in \mathsf{Passed}_i \setminus (\uparrow(\mathsf{AntiLosing}_i \cup \{\overline{v}\}))$. Thus, we conclude that any node which belongs to the set $\downarrow(\mathsf{Passed}_i \setminus (\uparrow(\mathsf{AntiLosing}_i \cup \{\overline{v}\})))$ is also in $\downarrow(\mathsf{AntiMaybe}_i) \setminus \{v\}$. Hence:

$$\downarrow(\mathsf{AntiMaybe}_{i+1}) \subseteq \downarrow(\mathsf{AntiMaybe}_i) \setminus \{v\}$$

Symmetrically, one can check that, at all times at the beginning of the loop $\mathsf{AntiMaybe}_i$ is included in Passed_i. Indeed, the only place where a node is added to $\mathsf{AntiMaybe}$ is line 27, where we have the guarantee that the node is indeed in Passed (see line 20). Hence, since a node cannot be in $\mathsf{AntiMaybe}$ and in $\mathsf{AntiLosing}$ at the same time, we conclude that $\downarrow(\mathsf{Passed}_i \setminus (\uparrow(\mathsf{AntiLosing}_i \cup \{\overline{v}\})))$ contains $\downarrow(\mathsf{AntiMaybe}_i) \setminus \{v\}$:

$$\downarrow(\mathsf{AntiMaybe}_{i+1}) \supseteq \downarrow(\mathsf{AntiMaybe}_i) \setminus \{v\}$$

Thus, $\downarrow(\mathsf{AntiMaybe}_{i+1}) = \downarrow(\mathsf{AntiMaybe}_i) \setminus \{\overline{v}\}$, as announced.

We conclude that, if v' (in the expression of Inv_i^2) is in $\downarrow(\mathsf{AntiMaybe}_i) \setminus \downarrow (\mathsf{AntiMaybe}_{i+1})$, then v' is necessarily the node that has been added to AntiLosing in line 37, and has triggered the re-evaluation of AntiMaybe in line 38. Thus, in line 39, $\mathsf{Depend}_i[v']$ is added to Waiting. We conclude this first point by considering two cases. When v' has originally been found in/inserted to $\downarrow(\mathsf{AntiMaybe})$ (say at step $j < i$), two possibilities might have occurred. The first case is that (v, v') has been inserted in $\mathsf{Depend}_j[v']$ (lines 17 or 26). Since the sets Depend never decrease, we conclude that $(v, v') \in \mathsf{Depend}_i[v']$, hence, by line 39, $(v, v') \in \mathsf{Waiting}_{i+1}$, and the invariant holds (✓). The second case is that there was some state $v_m \in \mathsf{AntiMaybe}_j$ s.t. $v_m \trianglerighteq v'$, and $(v, v') \in \mathsf{Depend}_j[v_m]$. Observe that it might be the case that v_m does not belong anymore to $\downarrow(\mathsf{AntiMaybe}_{i+1})$, because AntiMaybe might have been re-evaluated several times between step j and step i. However, if it is the case, either (v, v') has been inserted in Waiting at some point, and is still in $\mathsf{Waiting}_{i+1}$, hence the lemma holds (✓), or there is still $w \in \mathsf{AntiMaybe}_i$ s.t. $w \neq v'$, and $w \trianglerighteq v'$ and $(v, v') \in \mathsf{Depend}_i[w]$, because if it were not the case, it would mean that v' has been discovered as losing at a step before i, and line 38 would not have been reached during the ith iteration. In this latter case, since $w \neq v'$, we conclude that $w \in \mathsf{AntiMaybe}_{i+1}$, hence $(v, v') \in \mathsf{Depend}_{i+1}[\downarrow (\mathsf{AntiMaybe}_{i+1})]$, hence the invariant holds (✓).

2. Second, we consider the case where $(v, v') \in \mathsf{Waiting}_i$. If $(v, v') \in \mathsf{Waiting}_{i+1}$, then the invariant holds (✓). Otherwise, let us assume that $(v, v') \in \mathsf{Waiting}_i \setminus \mathsf{Waiting}_{i+1}$. This implies (v, v') has been popped (line 6) at the beginning of the $i + 1$th iteration, and there are three possible cases.

   (a) $(v, v') \in \mathsf{Depend}_{i+1}[\downarrow (\mathsf{AntiMaybe}_{i+1})]$ (because line 10, 15 or 17 has been executed) (✓)

   (b) $v' \in \mathsf{Passed}_{i+1} \setminus \mathsf{Passed}_i$ (because line 20 has been executed). Then, either $v' \in \uparrow(\mathsf{AntiLosing}_i)$ and thus the test in line 21 is false, line 33 is executed and $(v, v') \in \mathsf{Waiting}_{i+1}$ which contradicts our hypothesis that $(v, v') \notin \mathsf{Waiting}_{i+1}$ (✓); or (v, v') is added to Waiting in line 24, hence $(v, v') \in \mathsf{Waiting}_{i+1}$, which, again is not possible by hypothesis (✓); or line 27 is executed, hence $v' \in \downarrow(\mathsf{AntiMaybe}_{i+1})$ (✓).

   (c) (v, v') goes in the 'reevaluation' part (**else** block from line 34). Then, either there is no successor of v outside of $\uparrow (\mathsf{AntiLosing}_i)$ hence Losing^* evaluates to true, v is moved to AntiLosing (line 37) and removed from AntiMaybe (line 38), hence $v \notin \mathsf{AntiMaybe}_{i+1}$ (✓); or there exists a successor w of v which is not in $\uparrow (\mathsf{AntiLosing}_i)$ (i.e. Losing^* evaluates to false). In this latter case, either (v, w) has not been treated yet and thus $(v, w) \in \mathsf{Waiting}_{i+1}$ (✓), or $w \in \downarrow (\mathsf{AntiMaybe}_{i+1})$ (✓).

3. Finally, we consider the case where $(v, v') \in \mathsf{Depend}_i[\downarrow (\mathsf{AntiMaybe}_i)]$. Again, if $(v, v') \in \mathsf{Depend}_{i+1}[\downarrow (\mathsf{AntiMaybe}_{i+1})]$, then the invariant holds

(✓). Otherwise, we assume 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 37). One can assume that $w \neq v'$ because the case $w = v'$ is treated in case 1 above. Then, since $(v, v') \notin \mathsf{Depend}_{i+1}[\downarrow(\mathsf{AntiMaybe}_{i+1})]$, we conclude $(v, v') \in \mathsf{Depend}_i[w]$ because, w is the only state in Passed_i which is in $\downarrow(\mathsf{AntiMaybe}_i) \setminus \downarrow(\mathsf{AntiMaybe}_{i+1})$ (this holds since $\mathsf{AntiMaybe}_{i+1} = \lceil \mathsf{Passed}_{i+1} \setminus \mathsf{AntiLosing}_{i+1} \rceil$, 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_i. Hence, line 39 is executed and $\mathsf{Depend}_i[w]$ is added to $\mathsf{Waiting}$. We conclude that $(v, v') \in \mathsf{Waiting}_{i+1}$ and thus the invariant holds (✓) □

Proof (of invariant Inv_i^3). The proof of Inv_i^3 can be as for Inv_i^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 $\downarrow(\mathsf{AntiMaybe}_{i+1})$ (✓). □

Proof (of invariant Inv_i^4). At $i = 0$, $\mathsf{AntiLosing} = \emptyset$, hence the initialisation 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 23 and 37.

- (line 23) A state v' can be added in $\mathsf{AntiLosing}$ on line 23. In this case, the conditional on line 22 ensures that the state is in Bad, hence v' is in Losing.
- (line 37) A state v can be added in $\mathsf{AntiLosing}$ on line 37. In this case, the definition of $Losing^*$ and the conditional on line 36 ensures that v is in Losing. Indeed, if $v \in V_{\mathcal{S}}$ 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, $v \in V_{\mathcal{R}}$ 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. □

Proof (of invariant Inv_i^5). The invariant is easily established by checking lines 10, 15, 17, 26 and 41 which are the only lines where an edge is added to Depend. □

We are now ready to prove soundness of the algorithm. Recall the definition of the strategy $\hat{\sigma}_{\mathcal{G}}$we build upon termination of the algorithm (when $I \notin \uparrow(\mathsf{AntiLosing})$): for all $v \in \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$: $\hat{\sigma}_{\mathcal{G}}(v) = v'$ such that $v' \in \mathsf{Succ}(v) \cap \downarrow(\mathsf{AntiMaybe})$; and for all $v \in V_{\mathcal{S}} \setminus \mathsf{AntiMaybe}$, we let $\hat{\sigma}_{\mathcal{G}}(v) = \star$.

Proposition 4 (Soundness). *Upon termination of Algorithm 1:, either* $I \in \uparrow(\mathsf{AntiLosing})$ *and there is no winning strategy for* $\mathcal{S}$*, or* $\hat{\sigma}_{\mathcal{G}}$ *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 $\hat{\sigma}_{\mathcal{G}}$ is well-defined, i.e. for all $v \in \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$, 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 $v \in \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$ 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, $\hat{\sigma}_{\mathcal{G}}$ 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 $\hat{\sigma}_{\mathcal{G}}$, and $\mathbf{S} = \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$. The choice of the concretisation of $\hat{\sigma}_{\mathcal{G}}$ does not matter, because the hypothesis required by Proposition 1 are properties of $\sigma(v)$ for states $v \in \mathbf{S}$ only, and $\mathbf{S} = \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$ is exactly the support of $\hat{\sigma}_{\mathcal{G}}$. Let us show that σ respects the three hypothesis of Proposition 1, i.e. that: (i) $(\mathsf{AntiMaybe} \cap V_{\mathcal{S}} \cup \hat{\sigma}_{\mathcal{G}}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}})) \cap \mathsf{Bad} = \emptyset$; (ii) $I \in \downarrow^{\trianglerighteq}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}})$; and (iii) $\mathsf{Succ}((,) \hat{\sigma}_{\mathcal{G}}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}})) \subseteq \downarrow^{\trianglerighteq}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}})$.

(i) By definition of tba-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 22 which is false when a state is added to $\mathsf{AntiMaybe}$ (line 38). Moreover by definition of $\hat{\sigma}_{\mathcal{G}}$, $\hat{\sigma}_{\mathcal{G}}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}}) \subseteq \downarrow(\mathsf{AntiMaybe})$, hence (i) is satisfied.

(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 $\hat{\sigma}_{\mathcal{G}}$:

$$\hat{\sigma}_{\mathcal{G}}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}}) \subseteq \downarrow(\mathsf{AntiMaybe} \cap V_{\mathcal{R}})\,. \tag{1}$$

Let us first show that, upon termination in the case where $I \notin \uparrow(\mathsf{AntiLosing})$: $\mathsf{Succ}(\mathsf{AntiMaybe} \cap V_{\mathcal{R}}) \subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_{\mathcal{S}}$. This is a consequence of Inv_i^3. 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 $\hat{\sigma}_{\mathcal{G}}(w) \trianglerighteq \overline{v}$, therefore $\overline{v} \in \downarrow(\mathsf{AntiMaybe})$.

Hence, $\mathsf{Succ}\,(\mathsf{AntiMaybe} \cap V_{\mathcal{R}}) \subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_{\mathcal{S}}$. However, by definition of tba-simulation, and since no element from AntiMaybe are in Bad, this implies that:

$$\mathsf{Succ}\,(\downarrow(\mathsf{AntiMaybe} \cap V_{\mathcal{R}})) \subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_{\mathcal{S}}. \tag{2}$$

Finally, we observe that Succ is a monotonic function in the sense that $A \subseteq B$ implies $\mathsf{Succ}\,(A) \subseteq \mathsf{Succ}\,(B)$ for all sets A, B. Hence:

$$\begin{aligned} \mathsf{Succ}\,(\hat{\sigma}_{\mathcal{G}}(\mathsf{AntiMaybe} \cap V_{\mathcal{S}})) &\subseteq \mathsf{Succ}\,(\downarrow(\mathsf{AntiMaybe} \cap V_{\mathcal{R}})) && \text{by (1)} \\ &\subseteq \downarrow(\mathsf{AntiMaybe}) \cap V_{\mathcal{S}} && \text{by (2)} \end{aligned}$$

and thus (iii) holds.

Hence, by Proposition 1, the $\star$-strategy $\sigma|_{\mathbf{S}}$ (with $\mathbf{S} = \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$) is a $\trianglerighteq$-winning $\star$-strategy. It is easy to check that $\sigma|_{\mathbf{S}} = \hat{\sigma}_{\mathcal{G}}$, 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 $\mathcal{S}$ in the game. □

Remark on line 38. As already explained, all nodes that are inserted in AntiLosing are surely losing, while some nodes inserted in AntiMaybe might later be discovered as losing and moved to AntiLosing. In this case, the set AntiMaybe must be recomputed: this happens in line 38, and this computation can be costly (its complexity raises to $\theta(|\ \mathsf{Passed}\ | \times |\ \mathsf{AntiLosing}\ |)$, which is problematic as Passed contains all visited states). During our experiments (see Section 7), we found this re-computation of AntiMaybeto be the main bottleneck of our algorithm in term of running time, if line 38 is implemented naively. Instead, we propose to attach to each state v a list $\mathsf{Lt}[v]$ which contains the states removed from AntiMaybebecause of v, i.e. we let $\mathsf{Lt}[v] := \mathsf{Lt}[v] \cup \{v'\}$ every time we have $v' \in \mathsf{AntiMaybe}$, and we insert v s.t. $v \trianglerighteq v'$. Then, this information can be used to speed the computation of line 38: every time after a node v is removed from AntiMaybe (because v has been found losing), we consider all nodes $v_m \in \mathsf{Lt}[v]$, and: (i) either insert v_m in AntiMaybe if there is no $w \in \mathsf{AntiMaybe}$ s.t. $w \trianglerighteq v_m$; (ii) or insert v_m in AntiMaybe otherwise.

Why simulations are not sufficient. To close the discussion of our optimised version of OTFUR, let us explain why the notion tba-simulation is crucial for our optimisations. To this end, we exhibit two games equipped with simulation relations that are *not* tba-simulations, and show where our techniques fail. In Figure 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 Figure 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 $\hat{\sigma}_{\mathcal{G}}$ s.t. $\hat{\sigma}_{\mathcal{G}}(v) = v''$ and $\hat{\sigma}_{\mathcal{G}}(v') = \star$. Clearly this $\star$-strategy is not $\succeq$-winning (actually, there is no winning strategy).

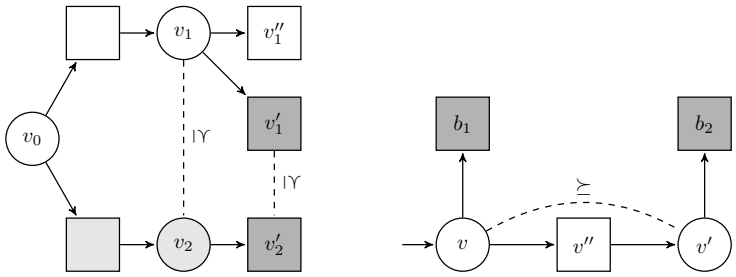


Figure 3: A simulation and the downward closure are not sufficient to apply Algorithm 1.

5.3. Finding tba-simulations

To apply our techniques, the game arena must be equipped with a *tba-simulation*. In many cases (see the three practical cases below), a *simulation relation* on the states of the game is already known, or can be easily defined. In general, not all simulation relations are tba-simulations, yet we can identify properties of the arena that yield this useful property. Intuitively, this occurs when Player $\mathcal{S}$ can always choose to play *the same set of actions* from all its states, and when 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 $\mathcal{G} = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ be a finite turn-based game and Σ a finite alphabet. A *labelling* of $\mathcal{G}$ is a function $\mathsf{lab} : E \to \Sigma$. For all states $v \in V_\mathcal{S} \cup V_\mathcal{R}$, and all $a \in \Sigma$, we let $\mathsf{Succ}_a(v) = \{v' \mid (v, v') \in E \wedge \mathsf{lab}(v, v') = a\}$. Then, $(\mathcal{G}, \mathsf{lab})$ is $\mathcal{S}$-*deterministic* iff there is a set of actions $\Sigma_\mathcal{S} \subseteq \Sigma$ s.t. for all $v \in V_\mathcal{S}$: (i) $|\mathsf{Succ}_a(v)| = 1$ for all $a \in \Sigma_\mathcal{S}$ and (ii) $|\mathsf{Succ}_a(v)| = 0$ for all $a \notin \Sigma_\mathcal{S}$. Moreover, a labelling lab is $\trianglerighteq$-*monotonic* (where $\trianglerighteq$ is a simulation relation on the states of $\mathcal{G}$) iff for all $v_1, v_2 \in V_\mathcal{S} \cup V_\mathcal{R}$ 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 $\mathcal{G} = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ be a finite turn-based game, let $\trianglerighteq$ be a simulation relation on $\mathcal{G}$ and let* lab *be a* $\trianglerighteq$-monotonic *labelling of $\mathcal{G}$. If $(\mathcal{G}, \mathsf{lab})$ is* $\mathcal{S}$-deterministic*, then $\trianglerighteq$ is a tba-simulation relation.*

Proof. Since $\trianglerighteq$ is a simulation relation, we only need to prove that for all $v_1 \in V_\mathcal{S}$, for all v_2 s.t. $v_1 \trianglerighteq v_2$, for all $v_1' \in \mathsf{Succ}(v_1)$, there is $v_2' \in V_\mathcal{R}$ s.t. $v_2' \in \mathsf{Succ}(v_2)$ and $v_1' \trianglerighteq v_2'$. Let $v_1, v_2 \in V_\mathcal{S}$ be s.t. $v_1 \trianglerighteq v_2$, and let v_1' be a state from $\mathsf{Succ}(v_1)$. Since $\mathcal{G}$ is $\mathcal{S}$-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 game $\mathcal{G}$ is labelled, $\mathcal{S}$-deterministic, equipped with a simulation relation $\trianglerighteq$ that can be computed directly from the description of the states[7]

[6] For example, in the urn-filling game (Figure 1), Player $\mathcal{S}$ can always choose between taking 1 or 2 balls, from all states where at least 2 balls are left.

[7] This means that one can decide whether $v \trianglerighteq v'$ from the encoding of v and v' and the set of pairs $\{(v, v') \mid v \trianglerighteq v'\}$ does not need to be stored explicitly.

and $\trianglerighteq$-monotonic, our approach can be applied out-of-the-box. In this case, Algorithm 1 yields, if it exists, a winning $\star$-strategy $\hat{\sigma}_{\mathcal{G}}$. We describe $\hat{\sigma}_{\mathcal{G}}$ by means of the set of pairs $(v, \mathsf{lab}(v, \hat{\sigma}_{\mathcal{G}}(v)))$ s.t. v is in the support of $\hat{\sigma}_{\mathcal{G}}$. That is, we store, for all v in the maximal antichain of winning reachable states, the *action* to be played from v instead of the *successor* $\hat{\sigma}_{\mathcal{G}}(v)$). Then, a controller implementing $\hat{\sigma}_{\mathcal{G}}$ works as follows: when the current state is v, the controller looks for a pair $(\overline{v}, a)$ with $\overline{v} \trianglerighteq v$, and executes a. Such a pair exists by $\mathcal{S}$-determinism (and respects $\trianglerighteq$-concretisation by $\trianglerighteq$-monotonicity). The time needed to find $\overline{v}$ depends only on the size of the antichain, that we expect to be small in practice.

6. Scheduling Games

In this section, we propose *Scheduling Games*, a framework for modelling and solving a real-time scheduling problem, namely the *feasibility problem of sporadic tasks on a multiprocessor system*. Roughly speaking, scheduling games are *safety games* played between a set of real-time tasks (the $\mathcal{R}$ player) that are scheduled one a multiprocessor platform by a real-time scheduler (the $\mathcal{S}$ player). Each task generates jobs that must be executed within a given deadline, and the duty of the scheduler is to avoid states where one of the jobs has missed its deadline (hence, the safety game). In such a setting, a *winning strategy* is a correct scheduling policy (or, simply, a correct schedule(r)). Since real-time schedulers must incur very little overhead in the execution of the system, which is very often implemented in an embedded system with scarce resources, having a succinct description of the winning strategy is of prime importance. Unfortunately, the combinatorial nature of the problem yields an explosion of the number of states of the arena. Hence, this scheduling problem is a perfect candidate to evaluate our approach in practice. Indeed, we will see in Section 7 that our approach performs much better on this problem than classical solutions in the literature. We start this section by informally describing the problem, then give the formal definitions.

Real-time Scheduling of sporadic tasks. We consider the problem of finding an *online scheduler* for a set of *sporadic tasks* on a *multiprocessor platform*. Here, *online* means that the scheduler has no *a priori* knowledge of the behaviour of the tasks, but must react, during the execution of the system to the requests made by the tasks. Intuitively, tasks release, recurrently, new *jobs* that must be executed by the system. Each job is equipped with a *computation time* C which is a natural number saying how much CPU time the job must consume to complete. Each job is also assigned a deadline D which is relative to its release time and says after how long the job must have been granted its C unit of CPU job lest it *misses its deadline*, a situation that the scheduler must avoid. Note that all jobs released by the same task have the same values for their C and D parameters. Finally, the rate at which jobs are released by the tasks is not fixed a priori, but a *minimal interarrival time* T is given.

The duty of the scheduler is to assign, at each time slot, the active jobs (those that are still waiting for CPU time) to m CPUs. Of course, in general,

their could be more active jobs than available CPUs. We assume that jobs can be freely migrated between CPUs, without incurring any delay; and that scheduler actions come at no cost.

An execution of the system alternates between actions of the tasks (that can submit new jobs to the system, provided they respect their interarrival time), and actions of the scheduler, that decides which jobs to assign to which CPU. When, the tasks have been assigned to the CPUs, some fixed time is allowed to elapse (depending on the granularity of the system), jobs that are assigned to the CPU consume one unit of CPU time, while the others do not. Then, the tasks can submit new jobs, etc. This clearly shows that the problem of finding a correct scheduler (the so-called *feasibility problem*) can be modelled by a *safety game*, where the $\mathcal{S}$ player is the scheduler, the $\mathcal{R}$ player is the coalition of the tasks, and the set of bad states is the set of all states where at least one job has missed its deadline.

Formally, we consider a set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of *sporadic tasks* to be scheduled on a multiprocessor platform with m identical processors. A sporadic task τ_i is a 3-tuple (T_i, D_i, C_i) where $T_i \in \mathbb{N}$ is its *minimum interarrival time* , $D_i \in \mathbb{N}$ is its *relative deadline* and $C_i \in \mathbb{N}$ is its *worst-case execution time*. As already explained, all jobs released by task τ_i will inherit its computation time C_i and its relative deadline D_i.

Remark that we impose no relation between T_i and D_i. So, in particular, if D_i is much larger than T_i, then it is possible that task τ_i has submitted several jobs to the system that are not completed yet. For instance, if $T_i = 2$ and $D_i = 5$, then, τ_i can release a job at instants 0, 2 and 4 reaching a total of three pending jobs at instant 4 without missing a deadline (yet). However, in this case, we assume that the system serves the jobs of the same task using a FIFO policy: it waits for the completion of the oldest active job (or, in other words, the one with the earliest deadline) before granting the CPU to the others. Nevertheless, in all states where the deadline has not been missed, the number of pending jobs of τ_i is always bounded by $\left\lceil \frac{D_i}{T_i} \right\rceil$.

Example 1. *Throughout this section, we consider a running example with four tasks and* 2 *CPUs, described in Table 1. In this table, for all* i*:* $U_i = C_i/T_i$ *denotes the system utilisation (or system load) of a given task.*

Figure 4 illustrates a prefix of an execution of this system. The vertical arrow on the top of the figure indicates which tasks submit jobs at each instant: τ_2 *and* τ_3 *submit at instant 0;* τ_1 *at instant 2;* τ_4 *at instant 3; and* τ_2 *submits a new job at instant* 5*, which is allowed since* $5 \geq T_2 = 4$ *time units have elapsed since its first submission at instant 0. The deadlines of the jobs submitted by* τ_2 *and* τ_1 *are illustrated by the thick arrows (where the origin is the instant of submission, and the arrow points to the instant of the deadline). The tasks execute on a platform with two CPUs. At instant 0, the scheduler schedules* τ_2 *on CPU 1 (until it completes in instant 3), and* τ_3 *on CPU 2. On instant 2, the scheduler preempts* τ_3 *from CPU 2 to schedule* τ_1 *instead, but schedules again* τ_3 *on CPU 2 at instant 3, hence* τ_1 *misses its deadline at instant* 4 *(marked by* $\times$*).*

Table 1: The task set used as our running example.

	C_i	T_i	D_i	U_i
τ_1	2	3	2	2/3
τ_2	3	4	3	3/4
τ_3	4	12	12	1/3
τ_4	3	12	12	1/4

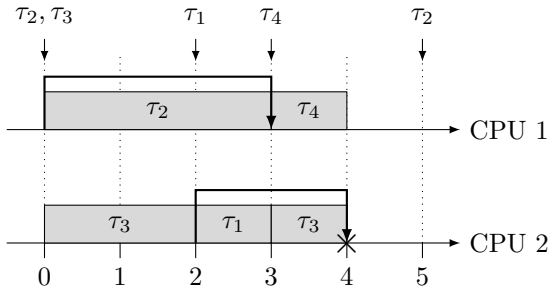


Figure 4: Illustration of a prefix of execution (for the task system in Table 1), where a deadline is missed at instant 4.

Related works. The feasibility problem for sporadic tasks is a well-studied problem in the real-time scheduling community. Apart for the very particular case where for each task the deadline corresponds to the period (implicit deadline systems $D_i = T_i \; \forall i$), where polynomial time feasibility tests exist, it exhibits a high complexity. For instance, even in the uniprocessor case, the feasibility problem of recurrent (periodic or sporadic) constrained deadline tasks is strongly co-NP hard [13]. For constrained/arbitrary deadline and sporadic task systems, several *necessary* or *sufficient* conditions have been identified for feasibility. However, these criteria are not sufficient to decide feasibility for all systems: there are some systems that do not meet any sufficient conditions (hence, are not surely feasible) and respect all identified necessary conditions (hence are not surely infeasible). We refer the reader to a survey by Davis and Burns [9] for more details on these conditions.

Since the sufficient and necessary conditions from the literature do not allow to answer feasibility on all systems, a so-called *exact* test (i.e. an algorithm to decide the problem) is highly desirable. To the best of our knowledge, the only exact test from the literature has been introduced by Bonifacci and Marchetti-Spaccamela [14]. They too, model the problem as a game and reduce the feasibility problem to finding a winning strategy (that corresponds to the scheduler) in this game. However, their algorithm is straightforward and no optimisations are proposed to make it applicable to realistic instances.

6.1. Game model

Let us now cast the informal definition of the game we have given above into our framework. Our definition of the game graph is inspired from the work of Baker and Cirinei [15] (like in [14], see also [10]) where they model a related problem, namely the sporadic tasks schedulability problem by means of a graph[8].

Set of states. In all states of the system, we store *two data* for each task τ_i: (i) the earliest arrival time $\text{NAT}\ (\tau_i)$ relative to the current instant; and (ii) the remaining processing time $\text{RCT}\ (\tau_i)$. Intuitively, at all times, $\text{NAT}\ (\tau_i)$ is the delay before τ_i can release a new job, while $\text{RCT}\ (\tau_i)$ is the amount of computation time the current job of τ_i still needs to complete. Observe that we keep, at all times, one pair $(\text{NAT}(\tau_i), \text{RCT}(\tau_i))$ per *task* τ_i. Implicitly, this information is related to the job (if any) of τ_i with the earliest deadline, which is the only one that can be scheduled (see our remark above). Formally:

Definition 2 (System states). *Let $\tau = \{\tau_1, \ldots, \tau_n\}$ be a set of sporadic tasks. A system state S of τ is a pair $(\text{NAT}_S, \text{RCT}_S)$ where: (i) NAT_S is a function from τ to $\mathbb{N}$ such that for all $\tau_i : \text{NAT}_S(\tau_i) \leq T_{max}$ with $T_{\max} = \max_i T_i$ and (ii) RCT_S is a function from τ to $\{0, 1, \ldots, C_{\max}\}$ with $C_{\max} = \max_i C_i$*

We denote by $\text{STATES}(\tau)$ the set of all system states of τ. Notice that, for each task set τ, the set $\text{STATES}(\tau)$ is infinite because there is no lower bound on $\text{NAT}_S(\tau_i)$. We will show later that we can restrict ourselves to a finite set of states to decide our problem. Let us now introduce several definitions related to the states. First, a task τ_i is said to be *active* in state S if it currently has a job that has not finished in S. Formally, the set of active tasks in S is $\text{ACTIVE}(S) = \{\tau_i \mid \text{RCT}_S(\tau_i) > 0\}$. To the contrary, an inactive task in S is called an *idle task*, i.e. τ_i is idle in S if and only if $\text{RCT}_S(\tau_i) = 0$. A task τ_i is *eligible* in a state S if it can submit a job from this configuration. Formally, the set of eligible tasks in the state S is: $\text{ELIGIBLE}(S) = \{\tau_i \mid \text{NAT}_S(\tau_i) \leq 0 \wedge \text{RCT}_S(\tau_i) = 0\}$. Finally, The LAXITY of a task τ_i in a system state S is defined by $\text{LAXITY}_S(\tau_i) = \text{NAT}_S(\tau_i) - (T_i - D_i) - \text{RCT}_S(\tau_i)$. Intuitively, the laxity of a task measures the amount of forthcoming CPU steps that the task could remain idle without taking the risk to miss its deadline. In particular, as we will see, states where the laxity is negative should be declared as losing states.

Example 2. *Let S_4 denote the state reached at instant 4 in Figure 4 with $\text{NAT}_{S_4}(\tau_4) = 11$; $\text{RCT}_{S_4}(\tau_4) = 2$. Then, one can check that $\text{ELIGIBLE}(S_4) = \{\tau_2\}$; $\text{ACTIVE}(S_4) = \{\tau_1, \tau_3, \tau_4\}$; and $\text{LAXITY}_{S_4}(\tau_1) = 1 - (3 - 2) - 1 = -1$. Thus, S_4 is a failure state.*

[8] In this problem, one is given a set of real-time sporadic tasks and a real-time scheduler, and one must determine whether this scheduler guarantees that no task ever miss a deadline. Hence, this problem can be regarded as a one-player game, or, in other words, as a reachability problem in a graph.

Now let us define the possible moves of both players. Let $S \in \textsc{States}(\tau)$ be a system state. We first define the possible moves of player $\mathcal{S}$, i.e. the scheduler. For all $\tau' \subseteq \textsc{Active}(S)$ s.t. $|\tau'| \leq m$ (i.e. τ' does not contain more tasks than the m available CPUs), we let $Succ_{\mathcal{S}}(S, \tau')$ be the (unique) state S' s.t. for all τ_i

$$\textsc{nat}_{S'}(\tau_i) = \begin{cases} \textsc{nat}_S(\tau_i) - 1 & \text{if } \textsc{rct}_S(\tau_i) > 0 \\ \max\{\textsc{nat}_S(\tau_i) - 1, 0\} & \text{otherwise} \end{cases}$$

and

$$\textsc{rct}_{S'}(\tau_i) = \begin{cases} \textsc{rct}_S(\tau_i) & \text{if } \tau_i \notin \tau' \\ \textsc{rct}_S(\tau_i) - 1 & \text{otherwise.} \end{cases}$$

Intuitively, each task $\tau_i \in \tau'$ is elected to be scheduled on a CPU, and consumes one CPU time unit. So we decrease the RCT's of those tasks only. We also decrease the NAT of all tasks, since one time unit has elapsed, which makes the potential next job arrival closer. Observe that when the RCT of the task is positive (i.e. the task is still active), we allow the NAT to go below zero. This will be used to model the fact that, when $T_i < D_i$ new jobs of τ_i can be submitted to the system even before the current job finishes, as will be clear later.

Let us now defined the possible moves of player $\mathcal{R}$, i.e. the tasks. Let $S \in \textsc{States}(\tau)$ be a system state, and let $\tau' \subseteq \textsc{Eligible}(S)$ be a set of eligible tasks. Then, we let $Succ_{\mathcal{R}}(S, \tau') \subseteq \textsc{States}(\tau)$ be the set of system states s.t. $S' \in Succ_{\mathcal{R}}(S, \tau')$ iff, for all τ_i:

$$\tau_i \notin \tau' \implies \textsc{nat}_{S'}(\tau_i) = \textsc{nat}_S(\tau_i)$$

and

$$\tau_i \in \tau' \implies \textsc{nat}_S(\tau_i) + T_i \leq \textsc{nat}_{S'}(\tau_i) \leq T_i$$

and

$$\textsc{rct}_{S'}(\tau_i) = \begin{cases} \textsc{rct}_S(\tau_i) & \text{if } \tau_i \notin \tau' \\ C_i & \text{otherwise.} \end{cases}$$

Example 3. *As an example, consider the $\mathcal{R}$ state $(S_3, \mathcal{R})$ reached at instant 3 in the execution in Figure 4. First, let's see how player $\mathcal{R}$ moves from S_3 to S'_3. We have* $\textsc{Eligible}(S_3) = \{\tau_2, \tau_4\}$. *One possible move for $\mathcal{R}$ is to have $\tau' = \{\tau_4\}$, i.e. that τ_4 submits a job, as in the figure. Then,* $\textsc{nat}_{S'_3}(\tau_4) = T_4 = 12$ *and* $\textsc{rct}_{S'_3}(\tau_4) = C_4 = 3$. *Meanwhile, the* NAT *and* RCT *of the other tasks are unchanged, i.e.* $\textsc{nat}_{S'_3}(\tau_1) = 3 - 1 = 2, \textsc{rct}_{S'_3}(\tau_1) = 2 - 1 = 1; \textsc{nat}_{S'_3}(\tau_2) = 4 - 3 = 1, \textsc{rct}_{S'_3}(\tau_2) = 3 - 3 = 0, \textsc{nat}_{S'_3}(\tau_3) = 12 - 3 = 9, \textsc{rct}_{S'_3}(\tau_3) = 4 - 2 = 2$. *Then player $\mathcal{S}$ can play from S'_3 to S_4. Note that* $\text{fl}(S'_3) = \{\tau_1, \tau_3, \tau_4\}$. *In the figure, Player $\mathcal{S}$ has decided to schedule $\tau' = \{\tau_3, \tau_4\}$. So, that* $\textsc{nat}_{S_4}(\tau_3) = \textsc{nat}_{S'_3}(\tau_3) - 1 = 8; \textsc{rct}_{S_4}(\tau_3) = \textsc{rct}_{S'_3}(\tau_3) - 1 = 1$ *and* $\textsc{nat}_{S_4}(\tau_4) = \textsc{nat}_{S'_3}(\tau_4) - 1 = 11; \textsc{rct}_{S_4}(\tau_4) = \textsc{rct}_{S'_3}(\tau_4) - 1 = 2$. *For $\tau_i \notin \tau'$, we have:* $\textsc{nat}_{S_4}(\tau_1) = \textsc{nat}_{S'_3}(\tau_1) - 1 = 1; \textsc{rct}_{S_4}(\tau_1) = \textsc{rct}_{S'_3}(\tau_1) = 1$ *and* $\textsc{nat}_{S_4}(\tau_2) = \textsc{nat}_{S'_3}(\tau_2) - 1 = 0; \textsc{rct}_{S_4}(\tau_2) = \textsc{rct}_{S'_3}(\tau_2) = 0$.

Let us now briefly explain why we allow the value of $\text{NAT}(\tau_i)$ to become negative. As explained above, it models the fact that a new job of τ_i has been submitted to the system while another job of the same task was not completed (this can happen without missing a deadline if $T_i < D_i$). Consider an example where $D_i = 5$, $T_i = 2$ and $C_i = 1$. When this task submits a first job, say, at instant $t = 0$, we have $\text{NAT}(\tau_i) = 2$ and $\text{RCT}(\tau_i) = 1$. Then, assume that the job idles during the next 2 time units, reaching a state where $\text{NAT}(\tau_i) = 0$ and still $\text{RCT}(\tau_i) = 1$, without missing a deadline which will occur at time $t = 5$. Thus, at time $t = 2$, τ_i could submit a new job which will anyway not be scheduled until the current one finishes. Then, assume the job gets scheduled, reaching a state where $\text{NAT}(\tau_i) = -1$ and $\text{RCT}(\tau_i) = 0$, just before the tasks can submit new jobs. The value $\text{NAT}(\tau_i) = -1$ records the fact that τ_i *could have* (because tasks are sporadic) submitted a job one time unit ago (at instant $t = 2$), but it can also submit the new job right now (at instant $t = 3$). This is why the possible moves in $Succ_{\mathcal{R}}$ allow the tasks to reach all the states where $\text{RCT}(\tau_i) = 1$ and $\text{NAT}(\tau_i) \in \{T_i - 1, T_i\} = \{1, 2\}$.

Finally, we need to define the notion of *failure states*, which are the losing states of the games. A state S is a *failure state* if there exists a task $\tau_i \in \text{ACTIVE}(S)$ such that $\text{LAXITY}(\tau_i) < 0$. From the definition of the laxity, it is easy to check (see [15, 10]) that those states are exactly the losing states, i.e. if a task has missed a deadline in a state S then S is a failure state; and if S is a failure state, then some task will miss its deadline whatever the scheduler does, because there isn't enough time left for the task to complete before the deadline.

We can now define the safety game in which we need to find a winning strategy in order to solve the feasibility problem for sporadic tasks on multiprocessor platforms. Given a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of sporadic tasks, and a number m of CPUs, we let $\mathcal{G}_{\tau,m}$ be the safety game $(V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ where:

- $V_{\mathcal{S}} = \{S \in \text{STATES}(\tau) \mid \forall 1 \leq i \leq n : \text{LAXITY}_S(\tau_i) \geq -1\} \times \{\mathcal{S}\}$ is the set of states controlled by the scheduler.
- $V_{\mathcal{R}} = \{S \in \text{STATES}(\tau) \mid \forall 1 \leq i \leq n : \text{LAXITY}_S(\tau_i) \geq -1\} \times \{\mathcal{R}\}$ is the set of states controlled by the environment.
- $E = E_{\mathcal{S}} \cup E_{\mathcal{R}}$ is the set of edges where:
  - $E_{\mathcal{S}}$ is the set of scheduler moves. It contains an edge $\big((S, \mathcal{S}), (S', \mathcal{R})\big)$ iff there is $\tau' \subseteq \text{ACTIVE}(S)$ s.t. $|\tau'| \leq m$ and $S' = Succ_{\mathcal{S}}(S, \tau')$. In this case, we sometimes abuse notations, and consider that this edge is labelled by τ', denoting it $\big((S, \mathcal{S}), \tau', (S', \mathcal{R})\big)$.
  - $E_{\mathcal{R}}$ is the set of tasks moves. It contains an edge $\big((S, \mathcal{R}), (S', \mathcal{S})\big)$ iff there exists $\tau' \subseteq \text{ELIGIBLE}(S)$ s.t. $v' \in Succ_{\mathcal{R}}(S, \tau')$. Again, we abuse notations and denote this edge by $\big((S, \mathcal{R}), \tau', (S', \mathcal{S})\big)$.
- $I = (S_0, \mathcal{R})$, where for all $1 \leq i \leq n$: $\text{RCT}_{S_0}(\tau_i) = \text{NAT}_{S_0}(\tau_i) = 0$ is the initial state.

- $\mathsf{Bad} = \{(S, \mathcal{P}) \in V_{\mathcal{S}} \cup V_{\mathcal{R}} \mid \exists \tau_i \in \textsc{Active}(S)$ such that $\textsc{Laxity}_S(\tau_i) < 0\}$, i.e. Bad is the set of failure states.

Observe that the set of states of the game is indeed *finite*, since we restrict ourselves to states where the laxity of all tasks is bounded below by -1. This implies that for all tasks τ_i and all states S: $\textsc{Laxity}_S(\tau_i) = \textsc{nat}_S(\tau_i) - (T_i - D_i) - \textsc{rct}_S(\tau_i) \geq -1$. Hence, $\textsc{nat}_S(\tau_i) \geq T_i - D_i + \textsc{rct}_S(\tau_i) - 1$. However, since $\textsc{rct}_S(\tau_i) \geq 0$ for all S and τ_i, we conclude that $\textsc{nat}_S(\tau_i) \geq T_i - D_i - 1$. This provides us with a lower bound on NAT that was not present in the definition of $\textsc{States}(\tau)$. We conclude that in the game $\mathcal{G}_{\tau,m}$, all states are of the form (S, P), with $P \in \{\mathcal{S}, \mathcal{R}\}$; and for all task τ_i: $\textsc{nat}_S(\tau_i) \in \{T_i - D_i - 1, \dots T_i\}$ and $\textsc{rct}_S(\tau_i) \in \{0, \dots, C_i\}$. Hence, $\mathcal{G}_{\tau,m}$ has at most $2 \times \big((D_{max} + 2) \times (C_{max} + 1)\big)^n = O\big((D_{max} \times C_{max})^n\big)$ states, and $\mathcal{G}_{\tau,m}$ is thus finite. The correctness of this restriction stems from the definition of the set of edges (see [10] for more detailed arguments): at each time step, the laxity can decrease of 1 at most. Hence, any play in the game reaching a Bad state must necessarily visit a (Bad) state where the laxity of at least one task is -1.

6.2. Turn-based alternating simulation for Scheduling Games

Now that we have defined the safety game we need to solve, we define a tba-simulation on the states of the game $\mathcal{G}_{\tau,m}$. Note that our definition of $\sqsupseteq$ extends the idle-task simulation relation defined in [10]. Our new relation is stronger and thus allows to optimise more aggressively the performance of our algorithm. From now on, we will often abuse notations, and write, for a state $v = (S, \mathcal{P})$ of a scheduling game $\mathcal{G}_{\tau,m}$, $\textsc{nat}_v$, $\textsc{rct}_v$, $\textsc{Eligible}(v)$, $\textsc{Active}(v)$, and $\textsc{Laxity}(v)$ instead of $\textsc{nat}_S$, $\textsc{rct}_S$, $\textsc{Eligible}(S)$, $\textsc{Active}(S)$, and $\textsc{Laxity}(S)$ respectively.

Definition 3 (Idle-ext task simulation). *Let τ be a set of sporadic tasks on a platform of m processors and let $\mathcal{G} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ be a scheduling game of τ on the platform. Then, the* idle-ext tasks preorder *$\sqsupseteq \subseteq V_{\mathcal{S}} \times V_{\mathcal{S}} \cup V_{\mathcal{R}} \times V_{\mathcal{R}}$ is a simulation relation; for all $v_1 = (S_1, \mathcal{P})$, $v_2 = (S_2, \mathcal{P}')$: $v_1 \sqsupseteq v_2$ iff $\mathcal{P} = \mathcal{P}'$ and, for all $\tau_i \in \tau$, the three following conditions hold: (i) $\textsc{rct}_{S_1}(\tau_i) \geq \textsc{rct}_{S_2}(\tau_i)$; and (ii) $\textsc{rct}_{S_2}(\tau_i) = 0$ implies $\textsc{rct}_{S_1}(\tau_i) = 0$; and (iii) $\textsc{nat}_{S_1}(\tau_i) \leq \textsc{nat}_{S_2}(\tau_i)$.*

The intuition of the idle-ext task simulation is that given $v_1 \sqsupseteq v_2$ then $\forall \tau_i \in \tau, \textsc{rct}_{v_1}(\tau_i) \geq \textsc{rct}_{v_2}(\tau_i)$ means that processors need more time for scheduling v_1 than v_2. Moreover, $\forall \tau_i \in \tau, \textsc{nat}_{v_1}(\tau_i) \leq \textsc{nat}_{v_2}(\tau_i)$ means that there are more tasks to schedule in v_1 than v_2. Therefore, it is more difficult to schedule v_1 than v_2. In other words, processors must work 'harder' for solving v_1 than v_2. In consequence, if v_2 is a losing state, then v_1 is a losing state as well. Similarly, v_1 is a winning state also implies v_2 is a winning state.

We start by proving two properties of this $\sqsupseteq$ relation before showing that it satisfies the definition of a tba-simulation. The properties are formalised by the next lemma, which relates the sets of active and eligible tasks of two states v_1 and v_2 s.t. $v_1 \sqsupseteq v_2$:

Lemma 2. *Let v_1 and v_2 be two states such that $v_1 \sqsupseteq v_2$, then: (i)* $\textsc{Active}(v_1) = \textsc{Active}(v_2)$*; and (ii)* $\textsc{Eligible}(v_1) \supseteq \textsc{Eligible}(v_2)$.

Proof. For the first item, we prove that $\textsc{Active}(v_1) \supseteq \textsc{Active}(v_2)$ and that $\textsc{Active}(v_2) \supseteq \textsc{Active}(v_1)$. First, given $\tau_i \in \textsc{Active}(v_2)$, then $\textsc{rct}_{v_2}(\tau_i) > 0$. As $\textsc{rct}_{v_1}(\tau_i) \geq \textsc{rct}_{v_2}(\tau_i)$, then $\textsc{rct}_{v_1}(\tau_i) > 0$, i.e $\tau_i \in \textsc{Active}(v_1)$. Hence, $\textsc{Active}(v_1) \supseteq \textsc{Active}(v_2)$.

Second, given $\tau_i \in \textsc{Active}(v_1)$, then $\textsc{rct}_{v_1}(\tau_i) > 0$. We assume that $\tau_i \notin \textsc{Active}(v_2)$, then $\textsc{rct}_{v_2}(\tau_i) = 0$. However, $\textsc{rct}_{v_2}(\tau_i) = 0$, implies $\textsc{rct}_{v_1}(\tau_i) = 0$, which is a contradiction. So, $\tau_i \in \textsc{Active}(v_2)$, and $\textsc{Active}(v_2) \supseteq \textsc{Active}(v_1)$. This proves the first item: $\textsc{Active}(v_1) = \textsc{Active}(v_2)$.

For the second item, given $\tau_i \in \textsc{Eligible}(v_2)$, then $\textsc{rct}_{v_2}(\tau_i) = 0$ and $\textsc{nat}_{v_2}(\tau_i) \leq 0$. We deduce that $\textsc{rct}_{v_1}(\tau_i) = 0$. Moreover $\textsc{nat}_{v_2}(\tau_i) \geq \textsc{nat}_{v_1}(\tau_i)$, then $\textsc{nat}_{v_1}(\tau_i) \leq 0$. In consequence $\tau_i \in \textsc{Eligible}(v_1)$. □

Theorem 5. *Given $\tau = \{\tau_1, \ldots, \tau_n\}$ a set of sporadic tasks and a multiprocessor platform with m processors. Let $\mathcal{G}_{\tau,m} = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ be the scheduling game for τ on a platform with m processors. Then, $\sqsupseteq$ is a turn-based alternating simulation for $\mathcal{G}_{\tau,m}$.*

Proof. To establish the property, we consider two states v_1 and v_2 in $V_{\mathcal{S}} \cup V_{\mathcal{R}}$ s.t. $v_1 \sqsupseteq v_2$, and we show that v_1 and v_2 respect Definition 1. We consider several cases. First, if $v_1 \in \mathsf{Bad}$, then Definition 1 holds trivially. From now on, let us assume that $v_1 \notin \mathsf{Bad}$, and let us consider separately the cases where $v_1 \in V_{\mathcal{S}}$ and $v_1 \in V_{\mathcal{R}}$.

First, assume $v_1 \in V_{\mathcal{S}}$. Let us show that for all $v_1' \in \mathit{Succ}_{\mathcal{S}}(v_1)$, there exists $v_2' \in \mathit{Succ}_{\mathcal{S}}(v_2)$ s.t. $v_1' \sqsupseteq v_2'$. Assume v_1' is s.t. $(v_1, \tau', v_1') \in E$, i.e. v_1' has been obtained from v_1 by scheduling $\tau' \subset \textsc{Active}(v_1)$. Let τ'' be the set of tasks $\tau'' = \{\tau_i \in \tau' \mid \textsc{rct}_{v_1}(\tau_i) = \textsc{rct}_{v_2}(\tau_i)\}$. Since $\tau'' \subseteq \tau' \subseteq \textsc{Active}(v_1)$, by definition, and since $\textsc{Active}(v_1) = \textsc{Active}(v_2)$ by Lemma 2, we conclude that $\tau'' \subseteq \textsc{Active}(v_2)$, and the state v_2' s.t. $(v_2, \tau'', v_2') \in E$ exists, by definition of $\mathit{Succ}_{\mathcal{S}}$. Let us show that $v_1' \sqsupseteq v_2'$. To do so, we first prove that the requested properties on $\textsc{rct}_{v_1'}$ and $\textsc{rct}_{v_2'}$ (i.e. points i and ii of Definition 3) hold for all task τ_i. We consider three cases depending on τ_i:

1. Let τ_i be a task s.t. $\tau_i \notin \tau'$ (hence, also, $\tau_i \notin \tau''$). In this case, by definition of $\mathit{Succ}_{\mathcal{S}}$, we have $\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_1}(\tau_i)$ and $\textsc{rct}_{v_2'}(\tau_i) = \textsc{rct}_{v_2}(\tau_i)$. Since $(\textsc{rct}_{v_1}(\tau_i) \geq \textsc{rct}_{v_2}(\tau_i))$ and $(\textsc{rct}_{v_2}(\tau_i) = 0$ implies $\textsc{rct}_{v_1}(\tau_i) = 0)$ because $v_1 \sqsupseteq v_2$, we conclude that the same holds for v_1' and v_2': $(\textsc{rct}_{v_1'}(\tau_i) \geq \textsc{rct}_{v_2'}(\tau_i))$ and $(\textsc{rct}_{v_2'}(\tau_i) = 0$ implies $\rightarrow \textsc{rct}_{v_1'}(\tau_i) = 0)$. Thus, for all $\tau_i \notin \tau'$, points i and ii of Definition 3 hold.

2. Let us now consider the case where $\tau_i \in \tau' \setminus \tau''$. In this case, we know that $\textsc{rct}_{v_1}(\tau_i) \neq \textsc{rct}_{v_2}(\tau_i)$, by definition of τ''. In addition, since $\textsc{rct}_{v_1}(\tau_i) \geq \textsc{rct}_{v_2}(\tau_i)$ (because $v_1 \sqsupseteq v_2$), we conclude that:

$$\textsc{rct}_{v_1}(\tau_i) > \textsc{rct}_{v_2}(\tau_i). \tag{3}$$

Then, by definition of $Succ_{\mathcal{S}}$ again, we have:

$$\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_1}(\tau_i) - 1 \tag{4}$$

$$\textsc{rct}_{v_2'}(\tau_i) = \textsc{rct}_{v_2}(\tau_i). \tag{5}$$

Combining (3), (4) and (5), we conclude that $\textsc{rct}_{v_1'}(\tau_i) \geq \textsc{rct}_{v_2'}(\tau_i)$. Moreover, $\textsc{rct}_{v_2'}(\tau_i)$ is necessarily different from 0 because otherwise, by (5), we would have $\textsc{rct}_{v_2}(\tau_i) = 0$, hence $\tau_i \notin \textsc{Active}(v_2)$, hence $\tau_i \notin \textsc{Active}(v_1)$ by Lemma 2; which is a contradiction with the hypothesis that $\tau_i \in \tau' \subseteq \textsc{Active}(v_1)$. Thus, the property on RCT holds. The same arguments as in the case where $\tau_i \notin \tau'$ can be applied to prove that $\textsc{nat}_{v_1'}(\tau_i) \leq \textsc{nat}_{v_2'}(\tau_i)$. Thus, for all $\tau_i \in \tau' \setminus \tau''$ points i and ii of Definition 3 hold.

3. Finally, let us consider the case where $\tau_i \in \tau'' \subseteq \tau'$. In this case, $\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_1}(\tau_i) - 1$ and $\textsc{rct}_{v_2'}(\tau_i) = \textsc{rct}_{v_2}(\tau_i) - 1$ by definition of $Succ_{\mathcal{S}}$. Moreover, since $\tau_i \in \tau''$, we have $\textsc{rct}_{v_1}(\tau_i) = \textsc{rct}_{v_2}(\tau_i)$. Hence, $\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_2'}(\tau_i)$, and thus points i and ii hold trivially.

Let us now turn our attention to $\textsc{nat}_{v_1'}$ and $\textsc{nat}_{v_2'}$, and let us show that point iii of Definition 3 holds. We consider two cases:

1. If $\textsc{rct}_{v_1}(\tau_i) > 0$, then $\tau_i \in \textsc{Active}(v_1)$. However, since $\textsc{Active}(v_1) = \textsc{Active}(v_2)$ by Lemma 2, we have $\textsc{rct}_{v_2}(\tau_i) > 0$. then, by definition of $Succ_{\mathcal{S}}$: $\textsc{nat}_{v_1'}(\tau_i) = \textsc{nat}_{v_1}(\tau_i) - 1$ and $\textsc{nat}_{v_2'}(\tau_i) = \textsc{nat}_{v_2}(\tau_i) - 1$. Since $v_1 \sqsupseteq v_2$, we have $\textsc{nat}_{v_1}(\tau_i) \leq \textsc{nat}_{v_2}(\tau_i)$. Hence, $\textsc{nat}_{v_1'}(\tau_i) \leq \textsc{nat}_{v_2'}(\tau_i)$

2. If $\textsc{rct}_{v_1}(\tau_i) = 0$, then, $\tau_i \notin \textsc{Active}(v_1)$. By Lemma 2 again, we conclude that $\tau_i \notin \textsc{Active}(v_2)$, hence $\textsc{rct}_{v_2}(\tau_i) = 0$. Then, by definition of $Succ_{\mathcal{S}}$, $\textsc{nat}_{v_1'}(\tau_i) = \max\{\textsc{nat}_{v_1}(\tau_i) - 1, 0\}$ and $\textsc{nat}_{v_2'}(\tau_i) = \max\{\textsc{nat}_{v_2}(\tau_i) - 1, 0\}$. Since $\textsc{nat}_{v_1}(\tau_i) \leq \textsc{nat}_{v_2}(\tau_i)$ as $v_1 \sqsupseteq v_2$, we conclude again that $\textsc{nat}_{v_1'}(\tau_i) \leq \textsc{nat}_{v_2'}(\tau_i)$.

Thus, when v_1 and v_2 are in $V_{\mathcal{S}}$, we have shown that there is v_2' with $(v_1, v_2') \in E$ s.t. $v_1' \sqsupseteq v_2'$.

Let us now consider the case where $v_1, v_2 \in V_{\mathcal{R}}$. Let us assume that $\tau'' \subseteq \textsc{Eligible}(v_2)$ is s.t. $(v_2, \tau'', v_2') \in E$, i.e. v_2' has been obtained from v_2 by letting the tasks in τ'' submit new jobs. Since $v_1 \sqsupseteq v_2$, we know, by Lemma 2 that $\textsc{Eligible}(v_1) \supseteq \textsc{Eligible}(v_2)$. Thus, the same set τ'' of tasks can submit jobs from v_1. Let v_1' be the state s.t. $(v_1, \tau'', v_1') \in E$, and where: (1) $\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_1}(\tau_i)$ for all $\tau_i \notin \tau''$; (2) $\textsc{rct}_{v_1'}(\tau_i) = C_i$ for all $\tau_i \in \tau''$; (3) $\textsc{nat}_{v_1'}(\tau_i) = \textsc{nat}_{v_1}(\tau_i)$ for all $\tau_i \notin \tau''$; and (4) Otherwise, $\textsc{nat}_{v_1}(\tau_i) + T_i = \textsc{nat}_{v_1'}(\tau_i)$ for all $\tau_i \in \tau''$. It is easy to check that such a v_1' exists by definition of $Succ_{\mathcal{R}}$. Let us show that $v_1' \sqsupseteq v_2'$. To this end, we show that all points of Definition 3 are satisfied for each task τ_i. We consider two cases:

1. First, let us assume that $\tau_i \notin \tau''$. Then, by definition of $Succ_{\mathcal{R}}$, $\textsc{rct}_{v_1'}(\tau_i) = \textsc{rct}_{v_1}(\tau_i)$, $\textsc{rct}_{v_2'}(\tau_i) = \textsc{rct}_{v_2}(\tau_i)$, $\textsc{nat}_{v_1'}(\tau_i) = \textsc{nat}_{v_1}(\tau_i)$ and $\textsc{nat}_{v_2'}(\tau_i) =$

$\text{NAT}_{v_2}(\tau_i)$. Hence, since $v_1 \sqsupseteq v_2$, points i and ii of Definition 3 hold on $\text{RCT}_{v'_1}(\tau_i)$ and $\text{RCT}_{v'_2}(\tau_i)$, and point iii holds on $\text{NAT}_{v'_1}(\tau_i)$ and $\text{NAT}_{v'_2}(\tau_i)$.

2. Second, let us assume that $\tau_i \in \tau''$. In this case, $\text{RCT}_{v'_1}(\tau_i) = \text{RCT}_{v'_2}(\tau_i) = C_i$, by definition of $Succ_{\mathcal{R}}$. So, points i and ii of Definition 3 hold trivially on $\text{RCT}_{v'_1}(\tau_i)$ and $\text{RCT}_{v'_2}(\tau_i)$. Moreover, $\text{NAT}_{v_2}(\tau_i) + T_i \leq \text{NAT}_{v'_2}(\tau_i) \leq T_i$ and $\text{NAT}_{v_1}(\tau_i) + T_i = \text{NAT}_{v'_1}(\tau_i)$, again by definition of $Succ_{\mathcal{R}}$. Since $\text{NAT}_{v_1}(\tau_i) \leq \text{NAT}_{v_2}(\tau_i)$, as $v_1 \sqsupseteq v_2$, we conclude that $\text{NAT}_{v'_1}(\tau_i) = \text{NAT}_{v_1}(\tau_i) + T_i = \text{NAT}v_2(\tau_i) + T_i \leq \text{NAT}_{v'_2}(\tau_i)$. Hence, $\text{NAT}_{v'_1}(\tau_i) \leq \text{NAT}_{v'_2}(\tau_i)$ and point iii of Definition 3 holds on $\text{NAT}_{v'_1}(\tau_i)$ and $\text{NAT}_{v'_2}(\tau_i)$

Thus, we shown that v'_1 defined as above is s.t. $v'_1 \sqsupseteq v'_2$ and $(v_1, v'_1) \in E$.

To conclude the proof, we need to show that, if $v_2 \in \mathsf{Bad}$, then $v_1 \in \mathsf{Bad}$ too. Let $\tau_i \in \text{fl}(v_2)$ be a task s.t. $\text{LAXITY}_{v_2}(\tau_i) < 0$. Such a task necessarily exists, by definition of Bad. Hence, $\text{NAT}_{v_2}(\tau_i) - (T_i - D_i) - \text{RCT}_{v_2}(\tau_i) < 0$. However, $v_1 \sqsupseteq v_2$ implies that, for all task τ_i, $\text{RCT}_{v_1}(\tau_i) \geq \text{RCT}_{v_2}(\tau_i)$ and $\text{NAT}_{v_1}(\tau_i) \leq \text{NAT}_{v_2}(\tau_i)$. Hence, $\text{NAT}_{v_2}(\tau_i) - (T_i - D_i) - \text{RCT}_{v_2}(\tau_i) < 0$ implies that $\text{NAT}_{v_1}(\tau_i) - (T_i - D_i) - \text{RCT}_{v_1}(\tau_i) < 0$, i.e. $\text{LAXITY}_{v_1}(\tau_i) < 0$. Thus, $v_1 \in \mathsf{Bad}$. This concludes the proof. □

7. Experimental Results

In order to assess the potential usefulness of our methods in practice, we have performed a series of experiments, that we report now. In this section, we denote by OTFUR-TBA our implementation of Algorithm 1 using the tba-simulation $\sqsupseteq$ defined in the previous section. For the sake of comparison, we have implemented two other algorithms. The first one is the exhaustive search (denoted ES) technique that consists in first building the whole game graph[9], then applying the attractor algorithm to compute the losing states. If the initial state of the graph is not losing, we can then extract a winning strategy that consists in selecting, in all winning nodes, any winning successor. In order to keep the comparison fair with OTFUR-TBA, the resulting strategy is then minimised wrt $\sqsupseteq$. Second, we have implemented the plain OTFUR algorithm, i.e. without our optimisations based on $\sqsupseteq$.

Our implementations are made in C++ using the STL. We run tests of a benchmark on a server equipped with Mac Pro (mid 2010) with OS X Yosemite, processor of 3.33 GHz, 6 core Intel xeon and memory of 32 GB 1333 MHz DDR3 ECC. Our programs were compiled with Apple Inc.'s distribution of g++ 4.2.1.

Experiments with T varying. Our first set of experiments consists in generating, randomly, sets of tasks (we keep *feasible sets only*), grouping the sets of tasks by values of T, the interarrival time. In these experiments, we always consider $n = 2$ CPUs and $m = 3$ tasks. For each i taking values in $\{5, 7, 9, 11, 13, 15, 17\}$,

[9]The size of the whole game graph also provides us with a measure of the difficulty of the instance.

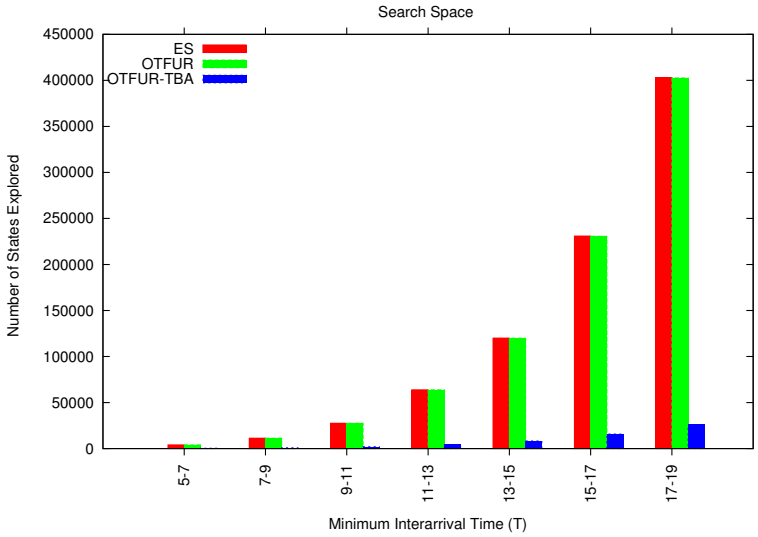


Figure 5: Search space explored by the three algorithms. The x-axes shows the range of T, corresponds to 300 game instances generated. Each column represents the average number of states explored for each range of T.

we generate a set, called $\tau^{[i,i+2]}$ of 300 tasks, s.t. in each set of $\tau^{[i,i+2]}$, the T parameters of all tasks ranges in the interval $[i, i+2]$. Thus, we produce a total of 2,100 task sets. To generate $\tau^{[i,i+2]}$, we proceed as follows. We first generate 20 tuples (T_1, T_2, T_3) giving the interarrival time of the three tasks. Then, from each tuple, we generate 15 task sets. To do so, we generate five task sets with a value[10] $U = 1$; 5 task sets with $U = 1.5$; and 5 task sets with $U = 2$. To achieve these values of U, we first compute, using the UUNIFAST algorithm [16], a tuple (U_1, U_2, U_3) s.t. $U = U_1 + U_2 + U_3$ has the desired value[11], then deduce suitable values of C_1, C_2 and C_3. The deadlines D_i are chosen between C_i and T_i, uniformly at random. Finally, we also discard all task sets that do not meet the following necessary condition for the feasibility of any sporadic arbitrary-deadline system: $\lambda_{sum}(\tau) \leq m$ and $\lambda_{max}(\tau) \leq 1$, where for all tasks τ_i: $\lambda_i = C_i / \min(D_i, T_i)$; $\lambda_{sum}(\tau) = (\sum_{\tau_i \in \tau} \lambda_i)$; and $\lambda_{max}(\tau) = (\max_{\tau_i \in \tau} \lambda_i)$.

For all these generated tasks sets, we run the three algorithms, and collect several metrics: the number of explored states (in the case of ES, it will be the number of states in the whole graph), the running time and the size of the winning strategy (i.e. the number of states v for which we need to store the value $\sigma(v)$, see Figure 1 for an example).

Figure 5 presents the average number of states explored in each set $\tau^{[i,i+2]}$ by the three algorithms. From this figure, we can already derive two obvious

[10] Recall that the utilisation factor $U = \sum_{i=1}^{n} U_i$ where each $U_i = \frac{C_i}{T_i}$

[11] We also discard the tuples where $U_i > 1$ for some task τ_i.

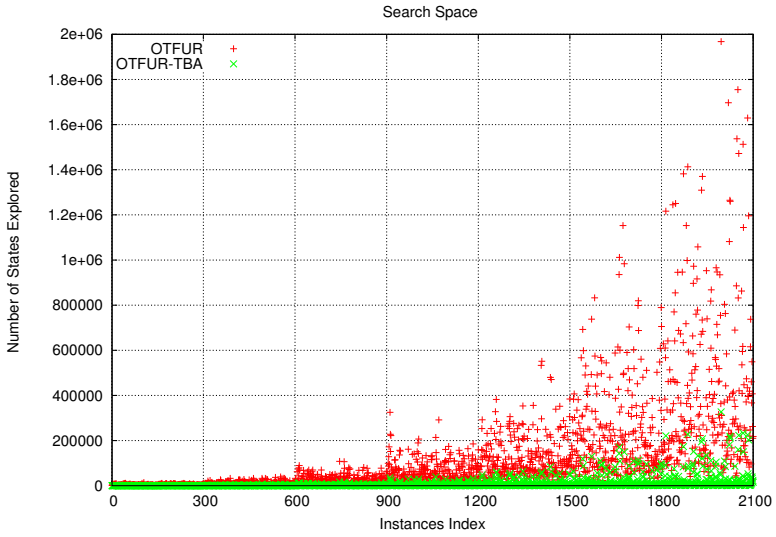


Figure 6: Comparison of the search space explored for each instance: OTFUR vs OTFUR-TBA.

conclusions. First, OTFUR-TBA clearly outperforms the two other algorithms ES and OTFUR. The average number of states explored by OTFUR-TBA is approximately 7% of the average number of states explored by ES and OTFUR, on all experiments. Second, it is interesting to remark that the number of states explored by ES and OTFUR are almost the same. This phenomenon can be explained by the following reasons. The main optimisation of OTFUR consists in *back propagating* the losing states, that is, when a state is declared *losing* during the forward exploration, it is surely losing, and this information is propagated towards the root, thereby allowing to declare the initial state losing as soon as possible. Since our main goal is to *compute schedulers*, we have considered only *feasible instances*, in our experiments, where initial state is always winning. Thus, although there can be reachable losing states in such games, the optimisations of OTFUR have limited impact in those cases. Moreover, the arenas of scheduling games contain many cycles (from all winning states, it is always possible to reach the initial state, as the tasks can always play in such a way that they do not submit jobs anymore). This prevents OTFUR from concluding quickly that the initial state is winning. From theses experiments, we can conclude that *positive* instances of the feasibility problem are apparently a class of worst cases for OTFUR, which further supports the importance of our contribution.

Since OTFUR and ES explore set of states that are very close to each other, we will henceforth present only a comparison between OTFUR-TBA and OTFUR (which can be regarded as the state-of-the-art algorithm). For the sake of completeness, figures presenting a comparison between OTFUR-TBA and ES are given in Appendix A.

Let us now turn our attention to Figure 6: it compares the number of states explored by OTFUR-TBA (in green) and OTFUR (in red). Each point on the graphics represent one task set. The indexes of the task sets have been assigned sequentially, following the possible values of T: the one with the highest indexes have their interarrival times in the interval $[i, i+2]$ with the largest i. As can be seen, even on such simple task sets (2 CPUs, 3 tasks), the number of states grows very quickly and peaks at nearly 2 millions. Clearly, OTFUR-TBA outperforms OTFUR (hence, also ES), and scales much better.

Next, we compare, in Figure 7, the sizes of the winning strategies computed by the two algorithms. The former figure compares the size of the winning strategy computed by OTFUR *before minimising*, to that computed by OTFUR-TBA (which is minimal by construction). The latter compares the winning strategies computed by OTFUR *after minimising* to that of OTFUR-TBA. These experiments further support our approach: while it is always possible to extract reasonably sized strategies from the strategies computed by OTFUR (and ES), this algorithm needs to explore redundant states, which might be prohibitive in practice. On the other hand, OTFUR-TBA keeps, at all times, sets that are minimal, and avoid exploring as much as possible redundant portions of the graph.

Finally, Figure 8 compares the running times of the algorithms—remember that, to ensure a fair comparison, the running time of OTFUR include a minimisation (wrt $\sqsupseteq$) of the computed strategy. As expected from the report on the number of explored states, OTFUR-TBA runs fastest and it takes only 20% the execution time the other two algorithms, in average.

Experiments with the number of tasks varying. Our goal with this second set of experiments is to determine the limit of all the algorithms, i.e. the biggest instance they are able to solve within 48h. We now let the number of tasks vary while the others parameters are fixed. The number of CPUs is set to 2, and we generate instances with $U = 2$ only. Again, the U_i parameters of the tasks was computed by the UUNIFAST algorithm. The minimum interarrival time T_i of each task was randomly generated in the range $[4, 6]$ and D_i in the rang $[3, 5]$. We generate only one instance (task set) for each value of n (number of tasks). The results are shown in Table 2.

As we can see in the table, both the search space and the running time explode very quickly. Furthermore, ES and OTFUR fail to solve the instance of 8 tasks because of the *out of memory* error. Meanwhile, our algorithm successes in solving this instance with a relatively small set AntiMaybe containing 26,648 states.

8. Conclusion and future works

We have introduced a framework to compute, efficiently, succinct strategies in safety (and reachability) games. We have demonstrated the potential of our approach by applying it to the *feasibility problem for sporadic tasks on multiprocessor platforms*, and important problem from the real-time scheduling

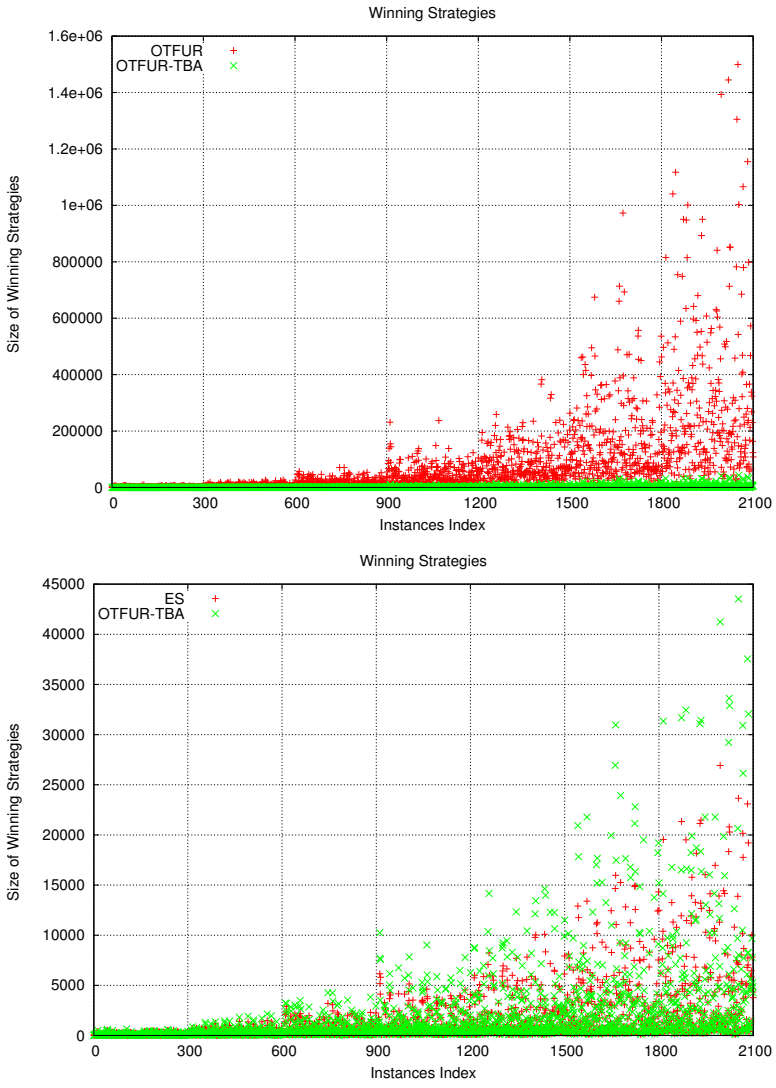


Figure 7: Comparison of the size of winning strategies: OTFUR vs OTFUR-TBA, for each game instance in the benchmark. On the top: the sizes for OTFUR are reported *before minimisation*. In the bottom: *after minimisation*.

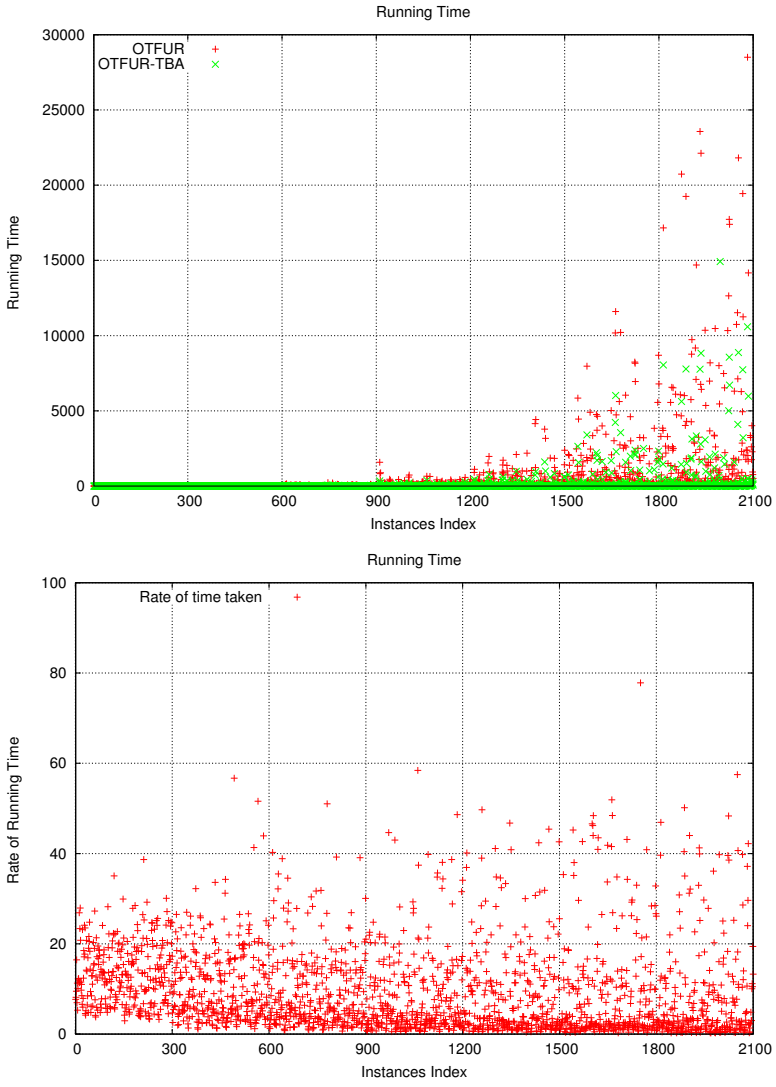


Figure 8: Comparison of the running times (top) and of the ratio of the running times (bottom): OTFUR vs OTFUR-TBA, for each game instance in the benchmark.

Table 2: Average number of states explored and time taken by the three algorithms for each number n of tasks.

n	ES		OTFUR		OTFUR-TBA	
	#States	Time (s.)	#States	Time (s.)	#State	Time (s.)
3	1,742	0.126	1,741	0.137	257	0.039
4	23,502	5.622	23,315	7.038	1,153	0.537
5	316,779	395.996	314,384	566.489	11,573	37.740
6	1,537,393	3,262.967	1,518,267	5,075.207	12,306	50.034
7	7,651,916	46,980.900	7,476,556	86,925.399	91,955	1,295.175
8	–	–	–	–	1,037,341	172,587.313

community. Our experiments show that our approach has the potential to push further the limit on the size of instances that can be handled in practice: while the number of states explored by ES and OTFUR are almost the same, OTFUR-TBA outperforms ES and OTFUR.

Yet, our algorithm has potentially other applications: we suggest two of them, that share the following characteristics, making our technique particularly appealing: (i) they have practical applications where an efficient implementation of the winning strategy is crucial; (ii) the arena of the game is not given explicitly and is at least exponential in the size of the problem instance; and (iii) they admit a natural tba-simulation $\trianglerighteq$, that can be computed directly from the descriptions of the states.

LTL realisability. roughly speaking, the realisability problem of LTL asks to compute a controller that enforces a specification given as an LTL formula. As already explained, Filiot, Jin and Raskin reduce [1] this problem to a safety game whose states are vectors of (bounded) natural numbers. They show that the partial order $\succeq$ where $v \succeq v'$ iff $v[i] \geq v'[i]$ for all coordinates i is a *simulation relation* and rely on it to define an efficient antichain algorithm (based on the OTFUR algorithm). Our technique generalises these results: Theorem 4 can be invoked to show that $\succeq$ is a *tba-simulation* and Algorithm 1 is the same as the antichain algorithm of [1], except for the third optimisation (see Section 5) which is not present in [1]. Thus, our results provide a general theory to explain the excellent performance reported in [1], and have the potential to improve it.

Determinisation of timed automata. Timed automata extend finite automata with a finite set of real-valued variables that are called clocks, whose value evolves with time elapsing, and that can be tested and reset when firing transitions [17] . They are a popular [18] model for real-time systems. One of the drawbacks of timed automata is that they *cannot be made deterministic in general*. Hence, 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 $\mathcal{G}_{\mathcal{A},(Y,M)}$ (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 strategy for Player $\mathcal{S}$. If the strategy is winning, then the approximation is an *exact* determinisation. Using Theorem 4, we can define a tba-simulation $\trianglerighteq_{\mathsf{det}}$ on the states of this game.

Future works. Besides these potential applications, we believe we could further enhance the performance of our prototype for the feasibility of sporadic tasks by using more efficient data structures. One possibility would be to use *Covering Sharing Trees* (CSTs) [19] to store efficiently sets of states. A version of the attractor algorithm using should also be investigated. Finally, another line of research would consists in devising an *ad hoc* data structure to store compactly sets of sets, in the spirit of *zones* and DBMs [20] for timed automata.

[1] E. Filiot, N. Jin, J. Raskin, Antichains and compositional algorithms for LTL synthesis, FMSD 39 (3) (2011) 261–296.

[2] V. Bonifaci, A. Marchetti-Spaccamela, Feasibility analysis of sporadic real-time multiprocessor task systems, in: Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10), Vol. 6347 of Lecture Notes in Computer Science, Springer, 2010, pp. 230–241.

[3] N. Bertrand, A. Stainer, T. Jéron, M. Krichen, A game approach to determinize timed automata, in: Proceedings of the 14th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'11), Vol. 6604 of Lecture Notes in Computer Science, Springer, 2011, pp. 245–259.

[4] C. Bouton, Nim, a game with a complete mathematical theory, Ann. Math., Ser. 2 3 (1902) 35–39.

[5] L. Doyen, J.-F. Raskin, Antichain algorithms for finite automata, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 6015 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 2–22. doi:10.1007/978-3-642-12002-2_2.
URL http://dx.doi.org/10.1007/978-3-642-12002-2_2

[6] G. Geeraerts, J. Goossens, M. Lindström, Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm, Real-Time Systems 49 (2) (2013) 171–218.

[7] R. Alur, T. A. Henzinger, O. Kupferman, M. Y. Vardi, Alternating refinement relations, in: Proceedings of the 9th International Conference on Concurrency Theory (CONCUR '98), Vol. 1466 of Lecture Notes in Computer Science, Springer, 1998, pp. 163–178.

[8] F. Cassez, A. David, E. Fleury, K. G. Larsen, D. Lime, Efficient on-the-fly algorithms for the analysis of timed games, in: (CONCUR'05), Vol. 3653 of LNCS, Springer, 2005, pp. 66–80.

[9] R. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Computing Surveys (CSUR) 43 (4) (2011) 35.

[10] G. Geeraerts, J. Goossens, M. Lindström, Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm, Real-Time Systems 49 (2) (2013) 171–218.

[11] D. Neider, Small strategies for safety games, in: T. Bultan, P.-A. Hsiung (Eds.), Automated Technology for Verification and Analysis, Vol. 6996 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 306–320.

[12] M. D. Wulf, L. Doyen, N. Maquet, J.-F. Raskin, Antichains: Alternative algorithms for ltl satisfiability and model-checking, in: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Vol. 4963 of Lecture Notes in Computer Science, Springer, 2008, pp. 63–77.

[13] P. Ekberg, W. Yi, Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly conp-complete, in: 27th Euromicro Conference on Real-Time Systems, IEEE, 2015, pp. 281–286.

[14] V. Bonifaci, A. Marchetti-Spaccamela, Feasibility analysis of sporadic real-time multiprocessor task systems, in: Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part II, Vol. 6347 of Lecture Notes in Computer Science, Springer, 2010, pp. 230–241.

[15] T. P. Baker, M. Cirinei, Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks, in: Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings, 2007, pp. 62–75.

[16] E. Bini, G. C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Systems 30 (1-2) (2005) 129–154. doi:10.1007/s11241-005-0507-9.
URL http://dx.doi.org/10.1007/s11241-005-0507-9

[17] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[18] J. Schmaltz, J. Tretmans, On conformance testing for timed systems, in: Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08), Vol. 5215 of Lecture Notes in Computer Science, Springer, 2008, pp. 250–264.

[19] G. Delzanno, J. Raskin, L. V. Begin, Covering sharing trees: a compact data structure for parameterized verification, STTT 5 (2-3) (2004) 268–297. doi:10.1007/s10009-003-0110-0.
URL http://dx.doi.org/10.1007/s10009-003-0110-0

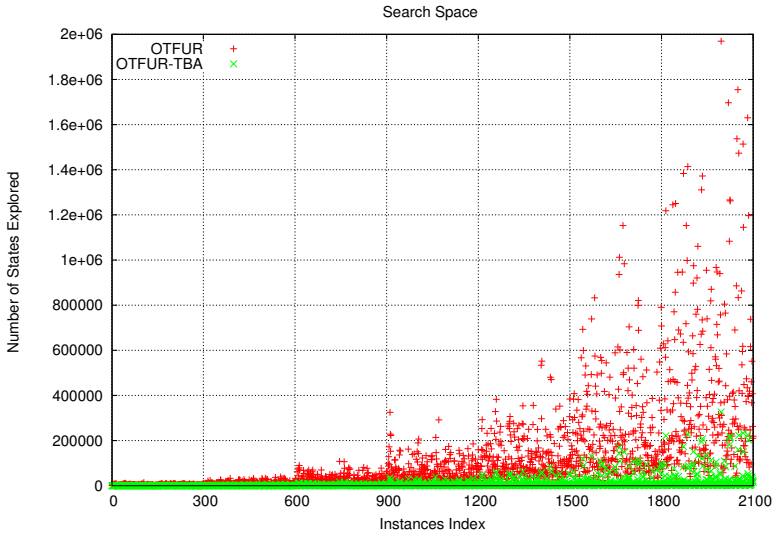


Figure A.9: Comparison of the search space explored for each instance: ES vs OTFUR-TBA.

[20] D. L. Dill, Timing assumptions and verification of finite-state concurrent systems, in: Springer (Ed.), Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings, Vol. 407 of Lecture Notes in Computer Science, 1989, pp. 197–212. doi:10.1007/3-540-52148-8_17.
URL http://dx.doi.org/10.1007/3-540-52148-8_17

Appendix A. Detailed comparison between es and otfur-tba

For the sake of completeness, this section compares the performances of ES and OTFUR-TBA. Since ES performs very close to OTFUR, the conclusions we can draw from these experiments are the same than the ones we have obtained when comparing OTFUR-TBA to OTFUR.

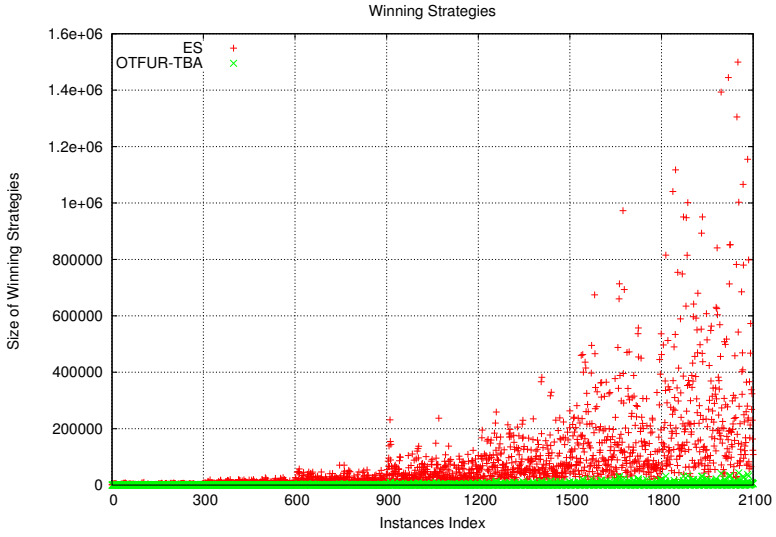


Figure A.10: Comparison of the size of winning strategies: ES vs OTFUR-TBA, for each game instance in the benchmark.

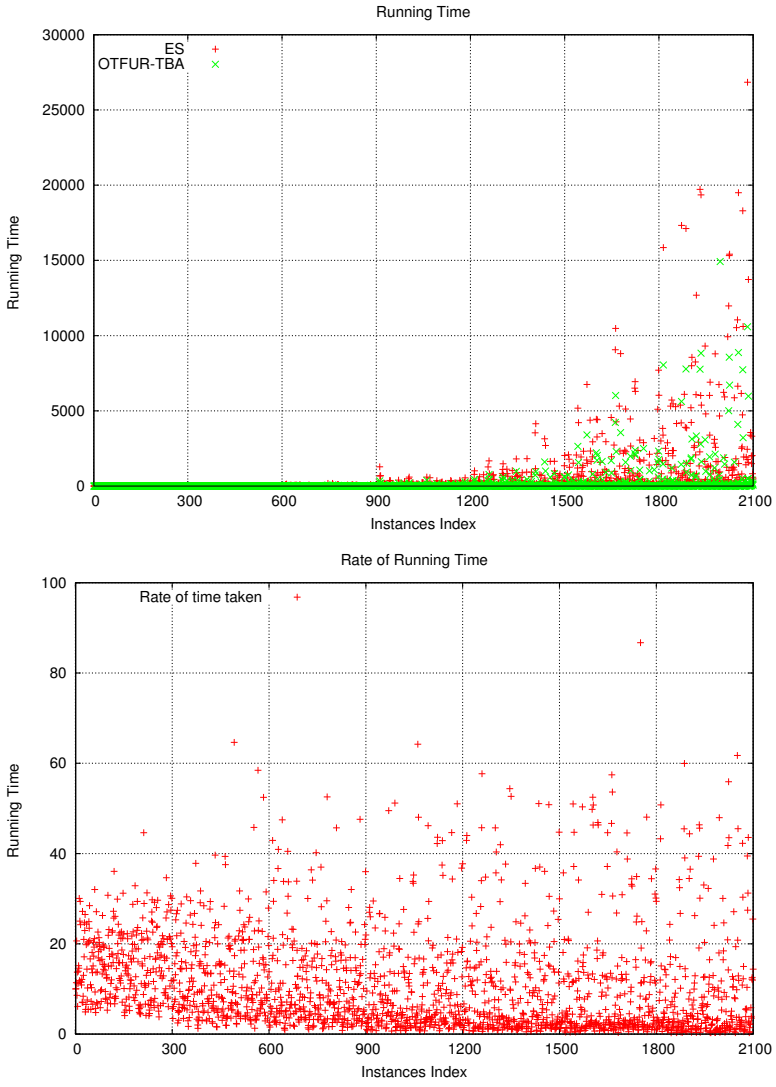


Figure A.11: Comparison of the running times (top) and of the ratio of the running times (bottom): ES vs OTFUR-TBA, for each game instance in the benchmark.