Synthesising Succinct Strategies in Safety and Reachability Games[⋆]

Gilles Geeraerts, Joël Goossens, and Amélie Stainer

Université libre de Bruxelles, Département d'Informatique, Brussels, 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. In particular, our techniques apply to the realisability problem of LTL [8], to the synthesis of real-time schedulers for multiprocessor platforms [4], and to the determinisation of timed automata [3] — three applications where the size of the game one needs to solve is at least exponential in the size of the problem description, and where succinct strategies are particularly crucial in practice.

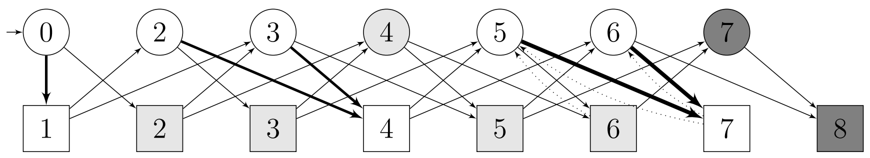
1 Introduction

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 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 (modeled 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 [8], real-time scheduler synthesis [4] or timed automata determinisation [3].

We consider, throughout the paper, a running example which is a variation of the well-known Nim game [5]. Initially, a heap of N balls is shared by the

[⋆] 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).



(a) Winning strategy

node	succ.	node	succ.	node	succ.	node	succ.
⓪	$\boxed{1}$	③	$\boxed{4}$	⑤	$\boxed{7}$	⑦	$\boxed{8}$
②	$\boxed{4}$	④	$\boxed{5}$	⑥	$\boxed{7}$		

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

node	succ.
⑤	$\boxed{7}$
⑥	$\boxed{7}$

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

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 looses the game if he is the last to play (i.e., the heap is empty after he has played). An arena modeling this game (for $N = 8$) is given in Fig. 1 (top), where $\mathcal{S}$-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 Fig. 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 $\mathsf{Bad} = \{⑦, \boxed{8}\}$, and we call *winning* all the states from which $\mathcal{S}$ can avoid Bad whatever $\mathcal{R}$ does. It is well-known [5] that a simple characterisation of the set of winning states[1] can be given. For each state v, let $\lambda(v)$ denote its label. Then, the winning states (in white in Fig 1) are all the $\mathcal{S}$-states v s.t. $\lambda(v) \mod 3 \neq 1$ plus all the $\mathcal{R}$-states v' s.t. $\lambda(v') \mod 3 = 1$.

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 Fig. 1 (a) for our running example), is not realistic because this table would have the size of the arena.

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

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 (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 [7]. 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 set of states that the algorithms need to maintain. They have been applied to several problems, such as LTL model-checking [7] or multi-processor schedulability [9] with remarkable performance improvements of several orders of magnitude.

In this paper, we introduce general antichain-based techniques for solving reachability and 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 [2]), 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 $\lambda(v) \mod 3 = \lambda(v') \mod 3$. Then, it is easy to see that the winning strategy of Fig. 1 (a) exhibits some kind of *monotonicity* wrt $\trianglerighteq_0$: 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 Fig. 1 (b). Observe that not all concretisations of this strategy are winning. For instance, playing $\boxed{3}$ from 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 Fig. 1 (b) 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 [6]). Our algorithm generalises the algorithm of Filiot et al. [8], 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 [8] ; 3. its proof of correctness is straightforward, and stems directly from the definition of tba-simulation. Finally, in Section 6, we show that our approach can be straightforwardly applied to the games one obtains in the three applications introduced above (LTL realisability, real-time feasibility and determinisation of timed automata) which demonstrates the wide applicability of our approach.

Note that, owing to lack of space, all proofs are to be found in the companion technical report [10]

2 Preliminaries

Turn-based finite 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 in Section 6. However, those 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 $G = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ 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 on to games: for a game $G = (\mathcal{A}, \mathsf{Bad})$, we let $\mathsf{Reach}(G, v) = \mathsf{Reach}(\mathcal{A}, v)$, $\mathsf{Reach}(G) = \mathsf{Reach}(\mathcal{A})$ and $\mathsf{Succ}(G, v) = \mathsf{Succ}(\mathcal{A}, v)$. 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 G_σ obtained from G by removing all the moves of $\mathcal{S}$ which are not chosen by σ, i.e. $\mathsf{Reach}(G_\sigma, v) \cap \mathsf{Bad} = \emptyset$, where $G_\sigma = (V_\mathcal{S}, V_\mathcal{R}, E_\sigma, I, \mathsf{Bad})$ 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 $G = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ iff it is winning in G for I.

Winning states and attractors. A state $v \in V$ in G is *winning* (for Player $\mathcal{S}$) iff there exists a strategy σ that is winning in 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 $G = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$, the sequence $(\mathsf{Attr}_i)_{i \geq 0}$ of attractors (of the Bad states) is defined as follows. $\mathsf{Attr}_0 =$

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 $G = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$. 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 $\trianglerighteq_0 = \{(v, v') \in V_{\mathcal{S}} \times V_{\mathcal{S}} \cup V_{\mathcal{R}} \times V_{\mathcal{R}} \mid \lambda(v) \geq \lambda(v') \text{ and } \lambda(v) \mod 3 = \lambda(v') \mod 3\}$ is a simulation relation compatible with $\mathsf{Bad} = \{\textcircled{7}, \boxed{8}\}$. Moreover, $\mathsf{Win} = \{v \in V_{\mathcal{S}} \mid \lambda(v) \mod 3 \neq 1\} \cup \{v \in V_{\mathcal{R}} \mid \lambda(v) \mod 3 = 1\}$ is downward closed for $\trianglerighteq_0$ and its

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

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

complement (the set of losing states), is upward closed. Finally, Win admits a single maximal antichain for $\trianglerighteq_0$: $\mathsf{MaxWin} = \{\boxed{7}, ⑥, ⑤\}$.

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 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 indifferent, for instance, when v is not reachable in G_σ. Thus, we want to reduce the number of states v s.t. $\sigma(v)$ is crucial to keep the system safe.

$\star$-strategies. We introduce the notion of $\star$-strategy to formalise this idea: a $\star$-strategy is a function $\hat{\sigma} : V_{\mathcal{S}} \mapsto V_{\mathcal{R}} \cup \{\star\}$, 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 $\mathsf{Supp}(\hat{\sigma})$.

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

Theorem 1. MinSizeStrat *is NP-complete.*

Thus, unless P=NP, there is no polynomial-time algorithm to compute a winning $\star$-strategy *of minimal size.* In most practical cases we are aware of, the situation is even worse, since the arena is not given explicitly. This is the case with the three problems we consider as applications (see Section 6), because they can be reduced to 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* [12,7,8]. In this line of research, the authors point out that, in practical applications (like those we identify in Section 6), system states exhibit some inherent *structure*, which is formalised by a *simulation relation* and can be exploited to improve the practical running time of the algorithms. In the present paper, we rely on the notion of *turn-based alternating simulation*, to define heuristics to (i) improve the running time of the algorithms to solve finite turn-based games and (ii) obtain succinct representations of strategies. This notion is adapted from [2].

Turn-based alternating simulations. Let $G = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ 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* G [2] (tba-simulation for short) iff for all v_1, v_2 s.t. $v_1 \trianglerighteq v_2$, either $v_1 \in \mathsf{Bad}$ or the three following conditions hold: (i) If $v_1 \in V_{\mathcal{S}}$, 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 (Fig. 1), $\trianglerighteq_0$ is a tba-simulation relation. Indeed, as we are going to see in Section 6, a simulation relation in a game where player $\mathcal{S}$ 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 Fig. 1 (b) and highlighted by bold arrows in Fig. 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 Fig. 1 (b) and represented by the boldest arrows in Fig. 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.

Obtaining $\trianglerighteq$-winning $\star$-strategies. The previous example clearly shows the kind of $\trianglerighteq$-winning $\star$-strategies we want to achieve: $\star$-strategies $\hat{\sigma}$ s.t. $\mathsf{Supp}(\hat{\sigma})$ is the maximal antichain of the winning states. In Section 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)

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

strategy, as shown by Proposition 1 hereunder. 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 $G = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$ be a finite turn-based game and $\trianglerighteq$ be a tba-simulation relation for G. Let σ be a strategy in 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.*

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

5 Efficient Computation of Succinct Winning Strategies

The original OTFUR algorithm. The *On-The-Fly algorithm for Untimed Reachability games* (OTFUR) algorithm [6] 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 [6] 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 optimized 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.

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

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

```
Data: A finite turn-based game $G = (V_{\mathcal{S}}, V_{\mathcal{R}}, E, I, \mathsf{Bad})$
if $I \in \mathsf{Bad}$ then return false;
Passed := $\{I\}$ ; Depend($I$) := $\varnothing$ ;
AntiMaybe := $\{I\}$ ; AntiLosing := $\{\}$ ;
Waiting := $\{(I, v') \mid v' \in \lfloor \mathsf{Succ}(I) \rfloor\}$ ;
while Waiting $\neq \varnothing \wedge I \notin\uparrow$ AntiLosing do
    $e = (v, v') := pop$(Waiting) ;
    if $v \notin\uparrow$ AntiLosing then
        if $v \in\downarrow$ AntiMaybe \ AntiMaybe then
            choose $v_m \in$ AntiMaybe s.t. $v_m \trianglerighteq v$ ;
            Depend[$v_m$] := Depend[$v_m$] $\cup \{e\}$ ;
        else
            if $v' \in\downarrow$ AntiMaybe then
                if $v' \notin$ AntiMaybe then
                    choose $v_m \in$ AntiMaybe s.t. $v_m \trianglerighteq v'$ ;
                    Depend[$v_m$] := Depend[$v_m$] $\cup \{e\}$ ;
            else
                if $v' \notin$ Passed then
                    Passed := Passed $\cup \{v'\}$ ;
                    if $v' \notin\uparrow$ AntiLosing then
                        if ($v' \in$ Bad) then
                            AntiLosing := $\lfloor$AntiLosing $\cup \{v'\}\rfloor$ ;
                            Waiting := Waiting $\cup \{e\}$ ; // reevaluation of $e$
                        else
                            Depend[$v'$] := $\{(v, v')\}$ ;
                            AntiMaybe := $\lceil$AntiMaybe $\cup \{v'\}\rceil$ ;
                            if $v \in V_{\mathcal{S}}$ then
                                Waiting := Waiting $\cup \{(v', v'') \mid v' \in \left\lfloor \mathsf{Succ}\left(v'\right) \right\rfloor\}$ ;
                            else
                                Waiting := Waiting $\cup \{(v', v'') \mid v' \in \left\lceil \mathsf{Succ}\left(v'\right) \right\rceil\}$ ;
                    else // reevaluation of $e$
                        Waiting := Waiting $\cup \{e\}$ ;
                else // reevaluation
                    Losing* := $v \in V_{\mathcal{S}} \wedge \bigwedge_{v'' \in \min(\mathsf{Succ}(v))} (v'' \in\uparrow \mathsf{AntiLosing})$ ;
                               $\vee\ v \in V_{\mathcal{R}} \wedge \bigvee_{v'' \in \max(\mathsf{Succ}(v))} (v'' \in\uparrow \mathsf{AntiLosing})$
                    if Losing* then
                        AntiLosing := $\lfloor$AntiLosing $\cup \{v\}\rfloor$ ;
                        AntiMaybe := $\lceil$Passed\ $\uparrow$(AntiLosing)$\rceil$ ;
                        // back propagation
                        Waiting := Waiting $\cup$ Depend[$v$] ;
                    else
                        if $\neg$Losing[$v'$] then Depend[$v'$] := Depend[$v'$] $\cup \{e\}$ ;
return $I \notin\uparrow$ AntiLosing
```

Optimised 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) $\mathsf{Waiting}$ that stores edges waiting to be explored; (ii) Passed that stores nodes that have already been explored; and (iii) $\mathsf{AntiLosing}$ and $\mathsf{AntiMaybe}$ 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 $\mathsf{Waiting}$, and checks whether exploring this edge can be postponed (line 7–15, see hereunder). Then, if v' has not been explored before (line 16), cannot be declared surely losing (line 18), and does not belong to Bad (line 19), it is explored (lines 23–28). When v' is found to be losing, e is put back in $\mathsf{Waiting}$ for back propagation (lines 21 or 30). The actual back-propagation is performed at lines 32–38 and triggered by an edge (v, v') s.t. $v' \in \mathsf{Passed}$. 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 3, 26 and 28). We consider maximal and minimal elements only when evaluating a node in line 32.
2. By Proposition 2, the set of winning states in the game is downward-closed, hence the set of losing states is upward-closed, and we store the set of states that are losing for sure as an antichain $\mathsf{AntiLosing}$ of minimal losing states.
3. Symmetrically, the set of *possibly winning states* is stored as an antichain $\mathsf{AntiMaybe}$ of maximal states. This set allows to postpone, and potentially avoid, the exploration of some states: assume some edge (v, v') has been popped from $\mathsf{Waiting}$. Before exploring it, we first check whether either v or v' belongs to $\downarrow(\mathsf{AntiMaybe})$ (see lines 7 and 11). If yes, there is $v_m \in \mathsf{AntiMaybe}$ s.t. $v_m \trianglerighteq v$ (resp. $v_m \trianglerighteq v'$), and the exploration of v (v') can be postponed. We store the edge (v, v') that we were about to explore in $\mathsf{Depend}[v_m]$, so that, if v_m is eventually declared losing (see line 36), (v, v') will be re-scheduled for exploration. Thus, the algorithm 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 [8]. Optimisation 3 is original and exploits more aggressively the notion of tba-simulation. It allows to keep at all times an antichain of potentially winning states, which is crucial to compute efficiently a winning $\star$-strategy. If, at the end of the execution, $I \not\in \uparrow(\mathsf{AntiLosing})$, we can extract from $\mathsf{AntiMaybe}$ a winning $\star$-strategy $\hat{\sigma}_G$ as follows. For all $v \in \mathsf{AntiMaybe} \cap V_{\mathcal{S}}$, we let $\hat{\sigma}_G(v) = v'$ such that $v' \in \mathsf{Succ}(v) \cap \downarrow(\mathsf{AntiMaybe})$. For all $v \in V_{\mathcal{S}} \setminus \mathsf{AntiMaybe}$, we let $\hat{\sigma}_G(v) = \star$. Symmetrically, if $I \in \uparrow(\mathsf{AntiLosing})$, there is no winning strategy for $\mathcal{S}$.

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

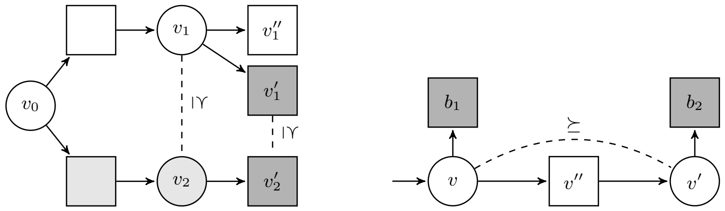


Fig. 2. A simulation and the downward closure are not sufficient to apply Algorithm 1

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

Why simulations are not sufficient. Let us exhibit two examples of games equipped with a simulation $\succeq$ which is not a tba-simulation, to show why tba-simulations are crucial for our optimisations. In Fig. 2 (left), $\mathsf{Bad} = \{v_1', v_2'\}$, and the set of winning states is not $\succeq$-downward closed (gray states are losing). In the game of Fig. 2 (right), $\mathsf{Bad} = \{b_1, b_2\}$ and Algorithm 1 does not develop the successors of v' (because $v \succeq v'$, and $v \in \mathsf{AntiMaybe}$ when first reaching v'). Instead, it computes a purportedly winning $\star$-strategy $\hat{\sigma}_G$ s.t. $\hat{\sigma}_G(v) = v''$ and $\hat{\sigma}_G(v') = \star$. Clearly this $\star$-strategy is not $\succeq$-winning (actually, there is no winning strategy).

6 Applications

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 $G = (V_\mathcal{S}, V_\mathcal{R}, E, I, \mathsf{Bad})$ be a finite turn-based game and Σ a finite alphabet. A *labeling* of 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, (G, 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 labeling lab is *$\trianglerighteq$-monotonic* (where $\trianglerighteq$ is a simulation relation on the states of G) iff for all $v_1, v_2 \in V_\mathcal{S} \cup V_\mathcal{R}$ 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:

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

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

Thus, when a game G is labeled, $\mathcal{S}$-deterministic, equipped with a simulation relation $\trianglerighteq$ that can be computed directly from the description of the states[7] 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 $\hat{\sigma}_G$. We describe $\hat{\sigma}_G$ by means of the set of pairs $(v, \mathsf{lab}(v, \hat{\sigma}_G(v)))$ s.t. v is in the support of $\hat{\sigma}_G$. 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* $\hat{\sigma}_G(v)$). Then, a controller implementing $\hat{\sigma}_G$ 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.

Three potential applications. Let us now describe very briefly three concrete problems to which our approach can be applied. They share the following characteristics, that make our technique particularly appealing: (i) they have practical applications where an efficient implementation of the winning strategy is crucial; (ii) the arena of the 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. The empirical evaluation of our approach is future work, except for the first application which has already been (partially) implemented in [8] with excellent performances.

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 [8] 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 [8], except for the third optimisation (see Section 5) which is not present in [8]. Thus, our results provide a general theory to explain the excellent performance reported in [8], and have the potential to improve it.

Multiprocessor real-time scheduler synthesis: this problem asks to compute a *correct scheduler* for a set of *sporadic tasks* running on a *platform of m identical CPUs*. A sporadic task (C, T, D) is a process that repeatedly creates *jobs*, s.t. each job creation (also called *request*) occurs at least T time units after the previous one. Each job models a computational payload. It needs at most C units of CPU time to complete, and must obtain them within a certain time frame of length D starting from the request (otherwise the job *misses* its

[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.

deadline). A scheduler is a function that assigns, at all times, jobs to available CPUs. It is *correct* iff it ensures that no job ever misses a deadline.

This problem can be reduced to a safety game [4] where the two players are the scheduler and the coalition of the tasks respectively. In this setting, a *winning* strategy for Player $\mathcal{S}$ is a *correct* scheduler. One can rely on Theorem 4 to show that the simulation relation $\succeq$ introduced in [9] (to solve a related real-time scheduling problem using antichain techniques) is a tba-simulation. An $\mathcal{S}$-deterministic and $\succeq$-monotonic labeling is obtained if we label moves of the environment by the set of tasks producing a request, and the scheduler moves by a total order on all the tasks, which is used as a priority function determining which tasks are scheduled for running.

Determinisation of timed automata: timed automata 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 [1]. They are a popular 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 $G_{\mathcal{A},(Y,M)}$ (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.

References

1. Alur, R., Dill, D.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
3. Bertrand, N., Stainer, A., Jéron, T., Krichen, M.: A game approach to determinize timed automata. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 245–259. Springer, Heidelberg (2011)
4. Bonifaci, V., Marchetti-Spaccamela, A.: Feasibility analysis of sporadic real-time multiprocessor task systems. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 230–241. Springer, Heidelberg (2010)
5. Bouton, C.: Nim, a game with a complete mathematical theory. Ann. Math. 3, 35–39 (1902)
6. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
7. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010)
8. Filiot, E., Jin, N., Raskin, J.: Antichains and compositional algorithms for LTL synthesis. FMSD 39(3), 261–296 (2011)

9. Geeraerts, G., Goossens, J., Lindström, M.: Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm. RTS 49(2), 171–218 (2013)
10. Geeraerts, G., Goossens, J., Stainer, A.: Computing succinct strategies in safety games. CoRR abs/1404.6228, `http://arxiv.org/abs/1404.6228`
11. Neider, D.: Small Strategies for Safety Games. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 306–320. Springer, Heidelberg (2011)
12. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: Alternative algorithms for LTL satisfiability and model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)